Efficient Reasoning about Data Trees via Integer Linear Programming

CLAIRE DAVID, Université Paris-Est
LEONID LIBKIN, University of Edinburgh
TONY TAN, University of Edinburgh

Data trees provide a standard abstraction of XML documents with data values: they are trees whose nodes, in addition to the usual labels, can carry labels from an infinite alphabet (data). Therefore, one is interested in decidable formalisms for reasoning about data trees. While some are known – such as the two-variable logic – they tend to be of very high complexity, and most decidability proofs are highly nontrivial. We are therefore interested in reasonable complexity formalisms as well as better techniques for proving decidability.

Here we show that many decidable formalisms for data trees are subsumed – fully or partially – by the power of tree automata together with set constraints and linear constraints on cardinalities of various sets of data values. All these constraints can be translated into instances of integer linear programming, giving us an NP upper bound on the complexity of the reasoning tasks. We prove that this bound, as well as the key encoding technique, remain very robust, and allow the addition of features such as counting of paths and patterns, and even a concise encoding of constraints, without increasing the complexity. The NP bound is tight, as we also show that the satisfiability of a single set constraint is already NP-hard.

We then relate our results to several reasoning tasks over XML documents, such as satisfiability of schemas and data dependencies and satisfiability of the two-variable logic. As a final contribution, we describe experimental results based on the implementation of some reasoning tasks using the SMT solver Z3.

Categories and Subject Descriptors: F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*; F.4.1 [**Mathematical logic and formal languages**]: Mathematical logic; G.1.6 [**Numerical Analysis**]: Optimization—*Integer programming*; H.2.1 [**Database Management**]: Logical Design—*Data Models*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: XML, tree languages, data values, Presburger arithmetic, reasoning, integer linear programming

ACM Reference Format:
David, C., Libkin, L., and Tan, T. 2012. Efficient Reasoning about Data Trees via Integer Linear Programming ACM 1, 1, Article 1 (December 2012), 27 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Traditional approaches to studying logics on trees use a finite alphabet for labeling tree nodes. The interest in such logics was reawakened by the development of XML as the

This work was supported by the FET-Open project FoX (Foundations of XML), grant agreement FP7-ICT-233599, and by EPSRC grant G049165. This work was done when the first author was at the University of Edinburgh.
Authors' addresses: C. David, Université Paris-Est; L. Libkin and T. Tan, University of Edinburgh. Emails: Claire.David@univ-mlv.fr, libkin@inf.ed.ac.uk, ttan@inf.ed.ac.uk.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

standard for data exchange on the Web. Logical formalisms provide the basis for query languages as well as for reasoning tasks, including many static analysis questions such as consistency of specifications, query optimization, and typing [Alon et al. 2003; Arenas et al. 2008; Fan and Libkin 2002; Figueira 2009; Genevés and Layaida 2006; Milo et al. 2003; Schwentick 2004].

The simplest abstraction of XML documents is ordered unranked finite trees whose nodes are labeled by letters from a finite alphabet [Neven 2002; Vianu 2001]. This abstraction works well for reasoning about structural properties, but real XML documents carry *data*, which cannot be captured by a finite alphabet. Thus, there has been a consistent interest in *data trees*, i.e., trees in which nodes carry both a label from a finite alphabet and a data value from an infinite domain [Bojanczyk et al. 2009; Bojanczyk et al. 2011; Bouyer et al. 2001; Demri and Lazic 2009; Neven et al. 2004; Kaminski and Tan 2008]. It seems natural to add at least the equality of data values to a logic over data trees. But while for finitely-labeled trees many logical formalisms are decidable by converting formulae to automata (e.g., the monadic second-order logic MSO), adding data-equality makes even FO (first-order logic) undecidable.

This explains why the search for decidable reasoning formalisms over data trees has been a common theme in XML research. Such a search has largely followed two routes. The first takes a specific XML reasoning task, or a set of similar tasks, and builds algorithms for them (see, e.g, [Arenas et al. 2008; Arenas and Libkin 2008; Björklund et al. 2008; Calvanese et al. 2009; Fan and Libkin 2002; Schwentick 2004; Figueira 2009]). The second attempts to find a sufficiently general logical formalism that is decidable, and can express some XML reasoning tasks of interest (see, e.g, [Demri and Lazic 2009; Bojanczyk et al. 2009; Jurdzinski and Lazic 2007]).

While both approaches have yielded many nontrivial and influential results, they are not completely satisfactory. The first approach gives us specialized algorithms for concrete problems, but no general tools. The second approach tends to produce extremely high complexity bounds, such as 4ExpTime, or even non-primitive-recursive. In addition, the proofs are usually highly nontrivial and are very hard to adapt to other reasoning tasks.

Instead we want a sufficiently general formalism – in fact, a family of formalisms, that are not extremely complicated to deal with, and at the same time give us acceptable complexity bounds. For reasoning tasks (as opposed to, say, query evaluation which we are used to in databases), acceptable complexity is often viewed as single-exponential [Robinson and Voronkov 2001], or better yet, NP. The latter is due to the fact that SAT solvers are now a practical tool for many static analysis problems [Malik and Zhang 2009].

The particular class of formalisms we deal with here is motivated by both concrete XML reasoning tasks and decidable logical formalisms. We now briefly describe those. One of the earliest reasoning problems studied in the XML context was the problem of reasoning about *keys and inclusion constraints*. An XML key says that for a given label a, the data value of an a-node (i.e., node labeled a) uniquely determines the node. An inclusion constraint says that every data value of an a-node will occur in a b-node as well. Such constraints are typical in databases, from which many XML documents are generated. The question is then whether they are consistent with the schema of an XML document, usually given as a tree automaton, or a DTD.

To see that such a problem may arise even with very simple specifications, consider, for instance, XML documents which allow element types a and b. Suppose we have a key constraint for data values in a-nodes, and an inclusion constraint from data values of a-nodes to data values of b-nodes. Clearly one can find documents satisfying these constraints. But now assume we have a DTD with a single rule $r \to aab$, where r is the root. Then this DTD is not compatible with the above constraints: since the a-nodes must carry two different data values, they cannot be included in the singleton set of data values of the only b-node.

In fact, [Arenas et al. 2008; Fan and Libkin 2002] have given a number of examples of naturally looking DTDs and sets of constraints that are inconsistent. The problem of checking consistency of DTDs with keys and inclusion constrains as described above is known to be solvable in NP [Fan and Libkin 2002].

On the logic side, there appear to be two main ideas leading to decidability. One starts with a temporal logic, and adds a limited memory for keeping and comparing data values. Examples include [Demri and Lazic 2009; Jurdzinski and Lazic 2007], but the logics, although decidable, have extremely high complexity (non-primitive-recursive). A different approach based on restricting the number of variables was followed by [Bojanczyk et al. 2009], which showed that FO^2, first-order logic with two variables, is decidable over data trees. In fact, even $\exists\mathrm{MSO}^2$, its extension with existential monadic second-order quantifiers, is decidable. The complexity drops to elementary but is still completely impractical: the decision procedure runs in 4ExpTime.

Our formalisms extend the specific constraints such as keys and inclusions, and yet come very close to subsuming the power of logics such as $\exists\mathrm{MSO}^2$, while permitting many properties which are not even definable in MSO – we will explain this more precisely in Section 6. To motivate the kind of constraints we use, let us restate keys and inclusion constraints in a slightly different way. For this, we need two new notations: $V(a)$ stands for the set of data values in a-labeled nodes, and $\#a$ is the number of a-nodes.

— A key simply states that $\#a = |V(a)|$. We view this as a *linear constraint*, and allow arbitrary linear constraints over the values $\#a$ and $|V(a)|$, for example, $|V(a)| \geq 2|V(b)| - \#c$.
— An inclusion constraint states that $V(a) \subseteq V(b)$, or, equivalently, $V(a) \cap \overline{V(b)} = \emptyset$. We generalize this to arbitrary *set-constraints* [Pacholski and Podelski 1997], stating that a Boolean combination of $V(a)$'s is either empty or nonempty.

We consider the problem of satisfiability of such constraints with respect to a schema declaration, given by an unranked tree automaton [Martens et al. 2007]. Or, formally: Given an unranked tree automaton $\mathcal{A}$ and a collection $\mathcal{C}$ of set and linear constraints, does there exist a tree t accepted by $\mathcal{A}$ that satisfies all the constraints in $\mathcal{C}$?

We prove that this problem, and several of its variations, are all NP-complete. The NP upper bounds are all established by reduction to instances of *integer linear programming*. In fact, the basic result, unlike many decidability proofs [Bojanczyk et al. 2009; Demri and Lazic 2009; Fan and Libkin 2002], is quite easy to establish. This opens a possibility of using efficient solvers for linear constraints to implement XML reasoning tasks. The lower bounds were already known [Fan and Libkin 2002], but we sharpen them significantly.

Our basic decidability result already subsumes not only reasoning about integrity constraints in XML, but also a very large fragment of $\exists\mathrm{MSO}^2$. These relationships will be made precise in Section 6. Note that even the decidability of the satisfiability problem does *not* follow from known results such as [Bojanczyk et al. 2009] which are restricted to fragments of MSO; in contrast, our formalism expresses many properties not definable in MSO.

In addition, our proof techniques come with extra benefits: we can easily extend reasoning tasks while retaining decidability. For instance, we can count not just the number of nodes labeled a, but also the number of nodes that satisfy some node tests of XPath: for instance, we can reason about the number of a-nodes that have a b-parent and a c-sibling. By extending the translation into integer linear programming, we obtain such extensions quite easily.

A more surprising extension is to *concisely represented* constraints. One way to reduce the size of the representation of linear constraints is to discard all zero entries from matrices defining them. This can shrink the size of the instance of the problem exponentially. A common phenomenon in complexity theory is that such a shrinking increases the complexity

by an exponent; this appears to suggest that the bound would be NExpTime. But we show that nonetheless the problem stays in NP.

In addition to proving the theoretical bounds, we would like to test the feasibility of our approach. There are several industrial-strength solvers for integer linear constraints that are used in program analysis, testing, and verification tasks. For our purposes we use the Z3 solver [de Moura and Bjørner 2008]. However, before we can use it, we need to solve one more problem, namely to encode automata with such constraints. It is well known that one can do it [Verma et al. 2005; Kopczynski and To 2010], and we use this fact in the decidability proof. However, for implementing our techniques, we cannot use it as a black box. Hence, we provide a self-contained translation from unranked tree automata to instances of linear programming. With that translation, we implement some of the reasoning tasks using Z3, and report initial promising results for DTDs with several hundred rules and constraints.

Remark. An extended abstract of this paper appeared in [David et al. 2011]. Compared to the conference version, there are three major additions:

— We have previously referred to [Fan and Libkin 2002] for lower bounds. However, that result required all possible constraints – automata, linear, and set – to achieve NP-hardness, and it was open whether all three were required. We have now solved this problem: we show in Subsection 4.2 that NP-hardness can be achieved with a single set constraint.
— We provide complete proofs in subsections 7.2 and 7.3 that were omitted in the proceedings version.
— Finally, after the conference publication, we have implemented some of the reasoning tasks using the Z3 solver. We describe initial experimental results in Section 9.

Organization. Section 2 presents the main definitions. In Section 3 we define the constraints and the problem we are interested in. In Section 4 we establish the basic result. An extension is presented in Section 5. In Section 6 we relate set and linear constraints to XML reasoning tasks and the logic $\exists\text{MSO}^2$ of [Bojanczyk et al. 2009]. We study the complexity of the problem in the case of concise representation of the constraints in Section 7. The translation from unranked tree automata to linear integer programming is provided in Section 8. We provide a preliminary report on our experimental results in Section 9. Concluding remarks are given in Section 10.

2. PRELIMINARIES

Trees and automata. We start with the standard definitions of unranked finite trees and logics and automata for them. An unranked finite tree domain is a prefix-closed finite subset D of $\mathbb{N}^*$ (words over $\mathbb{N}$) such that $u \cdot i \in D$ implies $u \cdot j \in D$ for all $j < i$ and $u \in \mathbb{N}^*$. Given a finite labeling alphabet Σ, a Σ-labeled unranked tree is a structure $\langle D, E_\downarrow, E_\rightarrow, \{a(\cdot)\}_{a\in\Sigma}\rangle$, where

— D is an unranked tree domain,
— $E_\downarrow$ is the child relation: $(u, u \cdot i) \in E_\downarrow$ for $u \cdot i \in D$,
— $E_\rightarrow$ is the next-sibling relation: $(u \cdot i, u \cdot (i+1)) \in E_\rightarrow$ for $u \cdot (i+1) \in D$, and
— the $a(\cdot)$'s are labeling predicates, i.e. for each node u, exactly one of $a(u)$, with $a \in \Sigma$, is true.

The label of the node u in t will be denoted by $\ell ab_t(u)$, and the domain D is denoted by $\mathsf{Dom}(t)$.

Let r be a designated symbol in Σ. We assume that the root of the tree (i.e., the empty word) is labeled r, and no other node is labeled r. (This is not a restriction as we can always put a new root with a given label.)

An *unranked tree automaton* [Comon et al. 2007; Thatcher 1967] over Σ-labeled trees is a tuple $\mathcal{A} = (Q, \Sigma, \delta, F)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and

$\delta : Q \times \Sigma \to 2^{(Q^*)}$ is a transition function; we require each $\delta(q, a)$ to be a regular language over Q for all $q \in Q$ and $a \in \Sigma$.

A run of $\mathcal{A}$ over a tree t is a function $\rho_{\mathcal{A}} : \mathsf{Dom}(t) \to Q$ such that for each node u with n children $u \cdot 0, \ldots, u \cdot (n-1)$, the word $\rho_{\mathcal{A}}(u \cdot 0) \cdots \rho_{\mathcal{A}}(u \cdot (n-1))$ is in the language $\delta(\rho_{\mathcal{A}}(u), \ell ab_t(u))$. For a leaf u labeled a, this means that u could be assigned a state q if and only if the empty word ϵ is in $\delta(q, a)$. A run is accepting if $\rho_{\mathcal{A}}(\epsilon) \in F$, i.e., if the root is assigned an accepting state. A tree t is accepted by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ on t. The set of all trees accepted by $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$.

Data trees. In a data tree, besides carrying a label from the finite alphabet Σ, each non-root node also carries a data value from some countably infinite data domain. To be concrete, we assume it to be $\mathbb{N}$. For a node u of a data tree t, labeled with a symbol $a \in \Sigma$, the assigned data value is denoted by $\mathrm{val}_t(u)$. We also denote the set of all data values assigned to a-nodes by $V_t(a)$. That is, $V_t(a) = \{\mathrm{val}_t(u) \mid \ell ab_t(u) = a \text{ and } u \in \mathsf{Dom}(t)\}$.

Integer linear programming (ILP) and Presburger formulae. Recall that an instance of integer linear programming consists of an $m \times k$ integer matrix $\mathbf{A}$ and a vector $\mathbf{b} \in \mathbb{Z}^m$. The question is whether there is a k-vector $\bar{v}$ over integers such that $\mathbf{A}\bar{v} \geq \mathbf{b}$.

The problem is well-known to be NP-complete. It is NP-hard even when entries are restricted to be 0's and 1's. Membership in NP follows from the fact that if $\mathbf{A}\bar{v} \geq \mathbf{b}$ has an integer solution, then there is one in which all entries are bounded by $(ak)^{p(m)}$, where a is the maximum absolute value that occurs in $\mathbf{A}$ and $\mathbf{b}$, and p is a polynomial [Papadimitriou 1981].

We also consider existential Presburger formulae, i.e., existential first-order formulae over the structure $\langle \mathbb{Z}, +, 0, 1, < \rangle$. Such formulae can always be converted to formulae of the form

$$\varphi(\bar{x}) \;=\; \exists \bar{y}\ \mathrm{PBC}(\mathbf{A}_i \bar{v}_i \geq \mathbf{b}_i), \tag{1}$$

where PBC means a positive Boolean combination, and each $\mathbf{A}_i \bar{v}_i \geq \mathbf{b}_i$ is an instance of integer linear programming with variables $\bar{v}_i$ coming from $\bar{x}, \bar{y}$. Indeed, each existential Presburger formula is of the form $\varphi(\bar{x}) = \exists \bar{y}\ \psi(\bar{x}, \bar{y})$, where ψ is quantifier-free, i.e., a Boolean combination of linear inequalities (both $>$ and $\geq$). Negations can be removed simply by changing the signs of coefficients, and strict inequalities $f(\bar{z}) > b$ can be replaced by conjunctions of $f(\bar{z}) - z' \geq b$ and $z' \geq 1$, where z' is a new existentially quantified variable.

Thus, whenever we refer to existential Presburger formulae, we assume that they are of the form (1). We also only work with non-negative integers for $\bar{x}$ and $\bar{y}$, so we always assume that all the conditions $x_j \geq 0, y_l \geq 0$ are included in formulae. However, it is to be noted that the entries in $\mathbf{A}_i$ and $\mathbf{b}_i$ can be negative.

Notice that we occasionally use conditions such as $x > 0$ or $x + y \leq b$, or $x = y$, but these are easily put in the form (1) by changing them to $x \geq 1$, or $-x - y \geq -b$, or the conjunction of $x \geq y$ and $y \geq x$, respectively.

The satisfiability of existential Presburger formulae is in NP. In [Papadimitriou 1981] it is showed for formulae of the form $\mathbf{A}\bar{v} \geq \mathbf{b}$. The proof immediately extends to our setting.

3. CONSTRAINTS AND THE SATISFIABILITY PROBLEM

In this section we give the precise definitions of set and linear constraints, and state the main satisfiability problem.

Set constraints. Recall that Σ is the labeling alphabet with a special symbol r for the root. *Data-terms* (or just terms) are given by the grammar

$$\tau := V(a) \mid \tau \cup \tau \mid \tau \cap \tau \mid \overline{\tau} \qquad \text{for } a \in \Sigma.$$

The semantics $[\![\tau]\!]_t$ is defined with respect to a data tree t:

— $[\![V(a)]\!]_t = V_t(a)$;
— $[\![\tau_1 \cap \tau_2]\!]_t = [\![\tau_1]\!]_t \cap [\![\tau_2]\!]_t$;
— $[\![\tau_1 \cup \tau_2]\!]_t = [\![\tau_1]\!]_t \cup [\![\tau_2]\!]_t$;
— $[\![\overline{\tau}]\!]_t = V_t - [\![\tau]\!]_t$;

where $V_t = \bigcup_{a\in\Sigma} V_t(a)$ is the set of data values found in the data tree t.

A *set constraint* is either $\tau = \emptyset$ or $\tau \neq \emptyset$, where τ is a term. A tree t satisfies $\tau = \emptyset$, written as $t \models \tau = \emptyset$, if and only if $[\![\tau]\!]_t = \emptyset$ (and likewise for $\tau \neq \emptyset$).

Note that set constraints $\tau_1 \subseteq \tau_2$ and $\tau_1 \subset \tau_2$ can be similarly defined, but they are easily expressible with the emptiness constraints. For example, $\tau_1 \subseteq \tau_2$ means that $\tau_1 \cap \overline{\tau_2} = \emptyset$, while $\tau_1 \subset \tau_2$ means that $\tau_1 \cap \overline{\tau_2} = \emptyset$ and $\tau_2 \cap \overline{\tau_1} \neq \emptyset$.

In particular, the inclusion constraint from the introduction is an example of a set constraint: to say that all data values of a-nodes occur as data values of b-nodes, we write $V(a) \cap \overline{V(b)} = \emptyset$.

Linear data constraints. Fix variables x_a for each $a \in \Sigma$ and z_S for each $S \subseteq \Sigma$. Linear data constraints are linear constraints over these variables.

The interpretation of x_a in a tree t is $\#a(t)$ – the number of a-nodes in t. The interpretation of z_S is the cardinality of the set

$$[S]_t = \bigcap_{a\in S} V_t(a) \cap \overline{\bigcup_{b\notin S} V_t(b)} = \bigcap_{a\in S} V_t(a) \cap \bigcap_{b\notin S} \overline{V_t(b)}.$$

That is, $[S]_t$ contains data values which are found among a-nodes for all $a \in S$ but which are not attached to any b-nodes for the label $b \notin S$. Note that the sets $[S]_t$'s are disjoint, and that

$$V_t(a) = \bigcup_{S \text{ such that } a\in S} [S]_t.$$

This gives us much more information than just the number of data values in a-nodes, which can be expressed as:

$$|V_t(a)| = \sum_{S \text{ such that } a\in S} z_S.$$

For instance, with such constraints we can reason about the data values that occur in a- and c-nodes but do not occur in b-nodes: the number of those is simply $\sum\{z_S \mid a, c \in S,\ b \notin S\}$.

Notice that key constraints from the introduction are examples of linear data constraints; that the data values of a-nodes form a key is that the number of a-nodes, which is x_a, is equal to the number of data values found in the a-nodes, which is $|V_t(a)|$. It is expressible by the linear constraint:

$$x_a = \sum_{S \text{ such that } a\in S} z_S.$$

We view linear data constraints as an instance of integer linear programming. If $\Sigma = \{a_1, \ldots, a_\ell\}$ and $S_1, \ldots, S_k$ is an enumeration of nonempty subsets of Σ (thus $k = 2^{|\Sigma|} - 1$), then a set of m linear data constraints is $\mathbf{A}\bar{v} \geq \mathbf{b}$, where $\mathbf{A}$ is an $m \times (\ell + k)$-matrix over $\mathbb{Z}$ and $\mathbf{b} \in \mathbb{Z}^m$. It is satisfied in a data tree t if it is true when $\bar{v}$ is interpreted as the vector

$$\big(\#a_1(t), \ldots, \#a_\ell(t),\ \big|[S_1]_t\big|,\ \ldots,\ \big|[S_k]_t\big|\big).$$

By data constraints, we mean either set constraints or linear data constraints.

Satisfiability problem. Let $\mathcal{C}$ denote a collection of set and linear data constraints. If a tree t satisfies all the constraints in $\mathcal{C}$, we write $t \models \mathcal{C}$. We study the following satisfiability problem.

PROBLEM:	$\mathrm{SAT}(\mathcal{A}, \mathcal{C})$
INPUT:	an unranked tree automaton $\mathcal{A}$,
	a collection $\mathcal{C}$ of set and linear data constraints
QUESTION:	is there a tree t accepted by $\mathcal{A}$ such that $t \models \mathcal{C}$?

The problem of consistency of XML constraints and schemas [Arenas et al. 2008; Fan and Libkin 2002] is a special instance of this problem. We shall later see that other problems related to XML constraints, as well as a large fragment of the two-variable logic can be formulated as special cases of $\mathrm{SAT}(\mathcal{A}, \mathcal{C})$. Moreover, $\mathrm{SAT}(\mathcal{A}, \mathcal{C})$ includes many instances that cannot even be formulated in MSO, which is the logic that typically subsumes XML reasoning tasks (for example, the linear constraint which states that $\#a(t) > 2 \cdot \#b(t)$ is not expressible in MSO, but is a simple linear data constraint $x_a - 2x_b > 0$).

4. DECIDING SATISFIABILITY

We now prove the decidability and the complexity of $\mathrm{SAT}(\mathcal{A}, \mathcal{C})$ problem. We assume the following way of measuring the size of the input:

— For the automaton $\mathcal{A}$, we take the size of the transition table, where each transition $\delta(q, a)$ is represented by an NFA (or by a regular expression, since an NFA can be computed from it in polynomial time).
— The size of each set constraint $\tau = \emptyset$, or $\tau \neq \emptyset$, is measured as the size of the parse-tree for the term τ.
— The size of the linear data constraints $\mathbf{A}\bar{v} \geq \mathbf{b}$ is the sum of sizes of $\mathbf{A}$ and $\mathbf{b}$, where the numbers are represented in binary.

The main decidability result is the following.

THEOREM 4.1. *The problem* $\mathrm{SAT}(\mathcal{A}, \mathcal{C})$ *is in* NP *(in fact,* NP*-complete).*

That the problem is NP-hard is already known [Fan and Libkin 2002], although we tighten the bound a lot in Subsection 4.2 (we show that satisfiability of a single set constraint is already NP-hard). The main contribution is an easy proof for the upper bound given in Subsection 4.1.

4.1. The proof of the NP-membership in Theorem 4.1

In this subsection we are going to present an NP-algorithm for $\mathrm{SAT}(\mathcal{A}, \mathcal{C})$.

Let $\Sigma = \{a_1, \ldots, a_\ell\}$ and $S_1, \ldots, S_k$ be the enumeration of non-empty subsets of Σ. We fix the vectors $\bar{x} = (x_1, \ldots, x_\ell)$ and $\bar{z} = (z_{S_1}, \ldots, z_{S_k})$.

We first show how to express set constraints in terms of linear data constraints. For this, we need the following notation. For a term τ over the alphabet Σ, we define a family $\mathbb{S}(\tau)$ of subsets of Σ as follows.

— If $\tau = V(a)$, then $\mathbb{S}(\tau) = \{S \mid a \in S \text{ and } S \subseteq \Sigma\}$.
— If $\tau = \overline{\tau}_1$, then $\mathbb{S}(\tau) = 2^\Sigma - \mathbb{S}(\tau_1)$.
— If $\tau = \tau_1 \star \tau_2$, then $\mathbb{S}(\tau) = \mathbb{S}(\tau_1) \star \mathbb{S}(\tau_2)$, where $\star$ is $\cap$ or $\cup$.

It follows immediately that for every data tree t, we have $[\![\tau]\!]_t = \bigcup_{S \in \mathbb{S}(\tau)} [S]_t$. Moreover, recall that the sets $[S]_t$'s are disjoint.

Thus, the set constraint $\tau = \emptyset$ can be expressed as a linear data constraint $\sum_{S \in \mathbb{S}(\tau)} z_S = 0$. Similarly, $\tau \neq \emptyset$ can be expressed as $\sum_{S \in \mathbb{S}(\tau)} z_S \geq 1$. Since the size of linear constraints is exponential in Σ, this transformation is polynomial in the size of the whole input.[1] Hence,

[1] In Section 7 when we look at the concise representations of the input, we will need a more refined technique for eliminating set constraints.

from now on, we can assume that the set $\mathcal{C}$ is of the form $\mathbf{A}(\bar{x}, \bar{z}) \geq \mathbf{b}$, and thus is given by a quantifier-free Presburger formula $\psi_{\mathcal{C}}(\bar{x}, \bar{z})$.

Next, we convert automata into linear constraints. In [Verma et al. 2005] it is shown that given a context free grammar G, whose terminals are $a_1, \ldots, a_\ell$, one can construct in polynomial time an existential Presburger formula $\varphi_G(x_1, \ldots, x_\ell)$ such that $\varphi_G(n_1, \ldots, n_\ell)$ holds if and only if there exists a word $w \in \mathcal{L}(G)$ such that $\#a_1(w) = n_1, \ldots, \#a_\ell(w) = n_\ell$, where $\#a_i(w)$ denotes the number of occurrences of a_i in the word w. Then, in [Kopczynski and To 2010] it is observed that the method can be extended to ranked tree automata. Since unranked tree automata can be easily converted to ranked tree automata with additional new symbol, we can construct the existential Presburger formula $\varphi_{\mathcal{A}}(x_1, \ldots, x_\ell)$ for unranked tree automaton $\mathcal{A}$, with one extra existential quantifier for the new symbol[2]. Hence, we have:

LEMMA 4.2. (See also Section 8.) *Given an unranked tree automaton $\mathcal{A}$, over alphabet $\Sigma = \{a_1, \ldots, a_\ell\}$, one can construct in polynomial time an existential Presburger formula $\varphi_{\mathcal{A}}(x_1, \ldots, x_\ell)$ such that if $t \in \mathcal{L}(\mathcal{A})$, then $\varphi_{\mathcal{A}}(\#a_1(t), \ldots, \#a_\ell(t))$ holds; and conversely, if $\varphi_{\mathcal{A}}(n_1, \ldots, n_\ell)$ holds, then there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that $\#a_1(t) = n_1, \ldots, \#a_\ell(t) = n_\ell$.*

Going back to the proof of Theorem 4.1, we introduce additional variables v_a for each $a \in \Sigma$. The intended meaning of v_a is the cardinality of $V_t(a)$. Let $\bar{v}$ be the vector $(v_{a_1}, \ldots, v_{a_\ell})$. We next define two formulae that ensure proper interaction between $\psi_{\mathcal{C}}$ and $\varphi_{\mathcal{A}}$. First,

$$\chi(\bar{v}, \bar{x}, \bar{z}) \;=\; \bigwedge_{a \in \Sigma} (v_a = \sum_{a \in S} z_S) \wedge (v_a \leq x_a)$$

states the expected conditions on these variables, given their intended interpretations. Second,

$$\chi'(\bar{v}, \bar{x}) \;=\; \bigwedge_{a \in \Sigma} (x_a = 0 \vee v_a > 0)$$

ensures that if a-nodes exist (i.e., $x_a > 0$), then at least one data value is attached to the a-nodes.

We now consider a Presburger formula $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{x}, \bar{z})$

$$\exists \bar{v} \; \big(\psi_{\mathcal{C}}(\bar{x}, \bar{z}) \wedge \varphi_{\mathcal{A}}(\bar{x}) \wedge \chi(\bar{v}, \bar{x}, \bar{z}) \wedge \chi'(\bar{x}, \bar{z})\big).$$

To convert $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{x}, \bar{z})$ into the form (1), we simply move all the existential quantifier in $\varphi_{\mathcal{A}}(\bar{x})$ to the front. Each atomic predicate inside $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{x}, \bar{z})$ can then be viewed as an instance of integer linear programming $\mathbf{A}\bar{y}_i \geq \mathbf{b}_i$.

LEMMA 4.3. *Given tuples of non-negative integers $\bar{n} = (n_a)_{a \in \Sigma}$ and $\bar{m} = (m_S)_{S \subseteq \Sigma}$, the formula $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{n}, \bar{m})$ holds if and only if there exists a data tree t accepted by $\mathcal{A}$ such that*

(1) $n_a = \#a(t)$ for each $a \in \Sigma$;
(2) $m_S = |[S]_t|$ for each $S \subseteq \Sigma$;
(3) $t \models \mathcal{C}$.

PROOF. The "if" part is immediate from the construction of $\Psi_{(\mathcal{A},\mathcal{C})}$. We prove the "only if" direction. Suppose $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{n}, \bar{m})$ holds. That is, there exists a witness $\bar{v}$ such that

$$\varphi_{\mathcal{C}}(\bar{n}, \bar{m}) \wedge \varphi_{\mathcal{A}}(\bar{n}) \wedge \chi(\bar{v}, \bar{n}, \bar{m}) \wedge \chi'(\bar{n}, \bar{m}) \text{ holds.}$$

Since $\varphi_{\mathcal{A}}$ holds, by Lemma 4.2, there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that $(\#a_1(t), \ldots, \#a_\ell(t)) = \bar{n}$.

[2]We shall present a more thorough construction in Section 8.

Now we show how to assign data values to the nodes in the tree t so that in the resulting data tree t' we have $m_S = |[S]_{t'}|$, for every $S \subseteq \Sigma$. Let $K = \sum_{S \subseteq \Sigma} m_S$. We use the set $\{1, \ldots, K\}$ for data values in the tree t'. Let

$$f : \{1, \ldots, K\} \mapsto 2^\Sigma - \emptyset$$

be a function satisfying $|f^{-1}(S)| = m_S$, for each $S \subseteq \Sigma$. The witness for $\bar{v}$ is $(\sum_{a_1 \in S} m_S, \ldots, \sum_{a_\ell \in S} m_S)$.

The data tree t' is obtained by letting $V_{t'}(a)$ be $\bigcup_{a \in S} f^{-1}(S)$. This is possible since $\chi(\bar{v}, \bar{n}, \bar{m})$ holds as $\sum_{a \in S} |f^{-1}(S)| = v_a \leq \#a(t) = n_a$. By definition of the function f, we obtain that $[S]_{t'} = f^{-1}(S)$, thus, $|[S]_{t'}| = m_s$, for each $S \subseteq \Sigma$. This proves the lemma. □

We now have an NP algorithm for SAT($\mathcal{A}, \mathcal{C}$): in polynomial time we construct the formula $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{x}, \bar{z})$ and then check for its satisfiability. It runs in NP, and Lemma 4.3 implies that it solves SAT($\mathcal{A}, \mathcal{C}$). □

We shall see in the next section that our algorithm for SAT($\mathcal{A}, \mathcal{C}$) gives some results obtained by using much harder techniques (such as reasoning about constraints in XML), and comes very close to giving us results obtained by *considerably much harder* techniques (like the results of [Bojanczyk et al. 2009]). Moreover, the structure of our proof leads to some extensions that otherwise would have been very hard to obtain.

Notice that extending the class of linear constraints by adding multiplication leads to the immediate loss of decidability, since Hilbert's 10th problem can be trivially encoded. On the other hand, the problem is decidable in NExpTime [Givan et al. 2002] if we extend linear constraints with *prequadratic* Diophantine equations, that is, Diophantine equations supplemented with constraints of the form $x_i \leq x_j x_k$.

4.2. The proof of the NP-hardness in Theorem 4.1

Recall that NP-hardness of the satisfiability problem already follows from [Fan and Libkin 2002], which, however, used all three kinds of constraints: automata (DTDs), linear constraints (keys), and set constraints (foreign keys). It was open whether NP-hardness can be established for less expressive constraints. Now we show that very little is needed for NP-hardness.

Proposition 4.4. *The problem of checking, for a set constraint $\tau \neq \emptyset$, whether there exists a tree t such that $t \models \tau \neq \emptyset$, is* NP*-hard.*

The proof goes via a reduction from the satisfiability of boolean formulae. We start by showing how to convert a boolean formula to a term. For a boolean formula φ with the variables $x_1, \ldots, x_\ell$, we define the term $\tau(\varphi)$ over the alphabet $\{a_1, \ldots, a_\ell\}$ as follows.

— If φ is x_i, then $\tau(\varphi)$ is $\overline{V(a_i)}$.
— If φ is $\neg\psi$, then $\tau(\varphi)$ is $\overline{\tau(\psi)}$.
— If φ is $\psi_1 \wedge \psi_2$, then $\tau(\varphi)$ is $\tau(\psi_1) \cap \tau(\psi_2)$.
— If φ is $\psi_1 \vee \psi_2$, then $\tau(\varphi)$ is $\tau(\psi_1) \cup \tau(\psi_2)$.

In the lemma below, we use the notion of $\mathbb{S}(\tau)$ defined in the beginning Subsection 4.1.

Lemma 4.5. *Let φ be a boolean formula over the variables $x_1, \ldots, x_\ell$, and $\xi : \{x_1, \ldots, x_\ell\} \mapsto \{\mathsf{true}, \mathsf{false}\}$ be a boolean assignment such that not all variables are assigned to* false*. The boolean formula φ evaluates to* true *under the assignment ξ if and only if the set $\{a_i \mid \xi(x_i) = \mathsf{true}\}$ is in $\mathbb{S}(\tau(\varphi))$.*

PROOF. The proof is by induction on the depth of the formula φ, denoted by $\text{depth}(\varphi)$.[3]

The basis is when $\text{depth}(\varphi) = 0$, that is, φ is x_i, for some $x_i \in \{x_1, \ldots, x_\ell\}$. Obviously x_i evaluates to true under the assignment ξ if and only if $\xi(x_i) = \mathsf{true}$. By definition of $\mathbb{S}$, the set $\mathbb{S}(\tau(x_i))$ contains the set $\{a_j \mid \xi(x_j) = \mathsf{true}\}$.

Assume now that Lemma 4.5 holds for every boolean formula of depth $< m$ and let φ be a boolean formula of depth m. There are three cases to consider.

(1) The formula φ is $\neg\psi$.
Let ξ be an assignment such that not all variables are assigned with false. If φ evaluates to true under the assignment ξ, then ψ evaluates to false under the assignment ξ. By the induction hypothesis, the set $\{a_j \mid \xi(x_j) = \mathsf{true}\}$ is not in $\mathbb{S}(\tau(\psi))$. By the definition, the set $\{a_j \mid \xi(x_j) = \mathsf{true}\}$ is in $\mathbb{S}(\tau(\varphi))$.
Vice versa, if the set $\{a_j \mid \xi(x_j) = \mathsf{true}\}$ is in $\mathbb{S}(\tau(\varphi))$, then it is not in $\mathbb{S}(\tau(\psi))$. By the induction hypothesis, the formula ψ evaluates to false under the assignment ξ. Therefore, the formula φ evaluates to true under the assignment ξ.

(2) The formula φ is $\psi_1 \wedge \psi_2$.
Let ξ be an assignment such that not all variables are assigned with false. If φ evaluates to true under the assignment ξ, then both formulae ψ_1 and ψ_2 evaluate to true under the assignment ξ. By the induction hypothesis, the set $\{a_j \mid \xi(x_j) = \mathsf{true}\}$ is in both $\mathbb{S}(\tau(\psi_1))$ and $\mathbb{S}(\tau(\psi_2))$. By the definition of $\mathbb{S}$, the set $\{a_j \mid \xi(x_j) = \mathsf{true}\}$ is in $\mathbb{S}(\tau(\varphi))$.
Vice versa, if the set $\{a_j \mid \xi(x_j) = \mathsf{true}\}$ is in $\mathbb{S}(\tau(\varphi))$, then it is in both $\mathbb{S}(\tau(\psi_1))$ and $\mathbb{S}(\tau(\psi_2))$. By the induction hypothesis, both formulae ψ_1 and ψ_2 evaluate to true under the assignment ξ. Therefore, the formula φ evaluates to true with the assignment ξ.

(3) The formula φ is $\psi_1 \vee \psi_2$.
This case can be proved in a similar manner as in the case 2, thus, we omit the proof.

This completes the proof of Lemma 4.5. □

The following lemma will immediately imply Proposition 4.4.

LEMMA 4.6. *For every boolean formula φ over the variables $x_1, \ldots, x_\ell$, the following holds. The formula $\varphi \wedge (x_1 \vee \cdots \vee x_\ell)$ is satisfiable if and only if there exists a data tree t such that $t \models \tau(\varphi) \neq \emptyset$.*

PROOF. We start with the "only if" part. Suppose that the boolean formula $\varphi \wedge (x_1 \vee \cdots \vee x_\ell)$ is satisfiable. Then there exists an assignment $\xi : \{x_1, \ldots, x_\ell\} \mapsto \{\mathsf{true}, \mathsf{false}\}$ such that not all the variables are assigned to false, and φ evaluates to true under ξ. By Lemma 4.5, the set $\{a_j \mid \xi(x_j) = \mathsf{true}\}$ belongs to $\mathbb{S}(\tau(\varphi))$, and since not all variables are assigned to false, it is not empty.

Consider the following tree t, whose root (which is labeled by the designated label r) has precisely $|\{a_j \mid \xi(x_j) = \mathsf{true}\}|$ children, and

— all these children are leaf nodes;
— there is exactly one node labeled with the label a, for each $a \in \{a_j \mid \xi(x_j) = \mathsf{true}\}$;
— all these leaf nodes have the same data value, say 1.

Now, for each set $S \subseteq \Sigma$, if $S = \{a_j \mid \xi(x_j) = \mathsf{true}\}$, then $[S]_t = \{1\}$. Otherwise, $[S]_t = \emptyset$. Since $[\![\tau(\varphi)]\!]_t = \bigcup_{S \in \mathbb{S}(\tau(\varphi))} [S]_t$, and the set $\{a_j \mid \xi(x_j) = \mathsf{true}\}$ is in $\mathbb{S}(\tau(\varphi))$, we have $[\![\tau(\varphi)]\!]_t \neq \emptyset$. Hence, $t \models \tau(\varphi) \neq \emptyset$.

Now we prove the "if" part. Suppose there exists a data tree t such that $t \models \tau(\varphi) \neq \emptyset$. This means that there is a set $S \in \mathbb{S}(\tau(\varphi))$ such that $[S]_t \neq \emptyset$. Consider the assignment $\xi : \{x_1, \ldots, x_\ell\} \mapsto (\mathsf{true}, \mathsf{false})$ where $\xi(x_i) = \mathsf{true}$ if and only if $a_i \in S$. The set S is not

[3]The depth of a boolean formula φ is defined as follows. $\text{depth}(x_i) = 0$; $\text{depth}(\neg\psi) = \text{depth}(\psi) + 1$; and $\text{depth}(\psi_1 \wedge \psi_2) = \text{depth}(\psi_1 \vee \psi_2) = \max(\text{depth}(\psi_1), \text{depth}(\psi_2)) + 1$.

empty, thus, not all the variables are assigned to false. Moreover, by Lemma 4.5, the boolean formula φ evaluates to true under the assignment ξ. Therefore, the formula $\varphi \wedge (x_1 \vee \cdots \vee x_\ell)$ is satisfiable. □

PROOF. (of Proposition 4.4) It is straightforward to establish the NP-hardness of the satisfiability of boolean formula of the form:

$$\varphi \wedge (x_1 \vee \cdots \vee x_\ell),$$

where $x_1, \ldots, x_\ell$ are all the variables in φ.

By Lemma 4.6, the satisfiability of $\varphi \wedge (x_1 \vee \cdots \vee x_\ell)$ can be reduced to the satisfiability of the set constraint $\tau(\varphi) \neq \emptyset$. Since the construction of $\tau(\varphi)$ can be done in linear time in the length of φ, Proposition 4.4 follows immediately. □

Remark 4.7. It is also true that given a term τ, deciding whether there exists a data tree t such that $t \models \tau = \emptyset$ is also NP-hard.

It can be established from the fact that there exists a data tree t such that $t \models \tau(\varphi) = \emptyset$ if and only if the boolean formula $\neg\varphi \wedge (x_1 \vee \cdots \vee x_\ell)$ is satisfiable. The proof is very similar to the proof above, and thus, omitted.

5. INCORPORATING COMPLEX PROPERTIES OF NODES

We now demonstrate how the simple structure of the proof allows us to obtain extensions for the main reasoning task almost effortlessly.

So far we were counting the numbers of nodes $\#a(t)$ – i.e., nodes labeled a. Checking whether a node is labeled a is a simple property expressed by a fixed MSO (in fact, by an atomic FO) formula with one free variable. We now show that we can count the number of nodes satisfying arbitrary fixed MSO formulae and use them in linear constraints.

More precisely, let $\pi(x)$ be an MSO formula with one free first-order variable in the usual vocabulary of unranked trees, that is, $E_\downarrow, E_\rightarrow$, and $a(\cdot)_{a\in\Sigma}$ for child and next-sibling edges and labeling predicates. Such a formula selects nodes in trees. We let $\#\pi(t)$ be the cardinality of the set of nodes in t that satisfy π.

Adding these variables to our system of constraints increases their expressiveness. For instance, we can state that every every a-labeled node is reachable by some XPath node expression $e(x)$. Indeed, unary MSO subsumes many XML formalisms, for example node expressions of XPath (or even conditional XPath [Marx 2005]). Thus, $e(x)$ can be viewed as an MSO formula, and then we can define $\pi(x) = e(x) \wedge a(x)$, stating that x is also labeled a. Then $\#\pi(t) = \#a(t)$ specifies the above constraint.

Using our proof, we can extend the decidability result to constraints that include counting the number of nodes output selected by unary MSO formulae. If $\Pi = \{\pi_1(x), \ldots, \pi_s(x)\}$ is a collection of such MSO formulae, then we refer to Π-linear constraints: these are linear constraints over the usual variables x_a's and z_S's, as well as w_{π_i}'s, interpreted as $\#\pi_i(t)$. We deal with the problem Π-SAT$(\mathcal{A}, \mathcal{C})$: its input is an automaton $\mathcal{A}$ and a collection $\mathcal{C}$ of set and Π-linear constraints, and the question is whether these are satisfiable.

Our proof immediately implies that the problem is decidable:

COROLLARY 5.1. *The problem* Π-SAT$(\mathcal{A}, \mathcal{C})$ *is decidable.*

PROOF. We can embed the formulae $\pi_1, \ldots, \pi_s$ into the automaton $\mathcal{A}$ and check the existence of a tree over the alphabet $\Sigma \times 2^\Pi$, where *(i)* its Σ projection is accepted by $\mathcal{A}$ and *(ii)* for each node labeled with $(a, P) \in \Sigma \times 2^\Pi$, a formula π is satisfied if and only if $\pi \in P$ is satisfied. The linear constraints in $\mathcal{C}$ over the variables x_a's and z_S's can be easily converted into the variables $x_{a,P}$'s and z_T, where $P \subseteq 2^\Pi$ and $T \subseteq (\Sigma \times 2^\Pi)$. □

The complexity of Π-SAT$(\mathcal{A}, \mathcal{C})$ depends on how the formulae $\pi_1, \ldots, \pi_s$ are given. If they are given as MSO formulae, then it is immediately known that the complexity is

non-elementary. But these formulae are also captured by the *query automata* of [Neven and Schwentick 2002]: these are automata that also select nodes in their accepting runs. When representing the formulae $\pi_1, \ldots, \pi_s$ with query automata, the complexity drops to NExpTime, and in some cases to NP.

Corollary 5.2. *If the formulae in* Π *are given as query automata, then* Π-SAT$(\mathcal{A}, \mathcal{C})$ *is decidable in* NExpTime. *Moreover,* Π-SAT$(\mathcal{A}, \mathcal{C})$ *is in* NP *if* Π *is fixed.*

Proof. The NExpTime upper bound is straightforward, as the non-elementary blow-up for SAT$(\mathcal{A}, \mathcal{C})$ occurs in translating the MSO formulae to query automata. Moreover, the blow-up occurs when moving from the alphabet Σ to $\Sigma \times 2^{\Pi}$. Thus, if Π is fixed, then the complexity remains NP. □

While converting from MSO to query automata is non-elementary, for some other formalisms that complexity is much lower: for example, [Libkin and Sirangelo 2010] shows how to convert conditional-XPath to query automata in single-exponential time.

6. COMPARISON WITH OTHER FORMALISMS

We now explain how the satisfiability problem SAT$(\mathcal{A}, \mathcal{C})$ relates to reasoning tasks for XML with data.

6.1. XML constraints

As we already noticed, keys and inclusion constraints, studied extensively in the XML context (and included in the standards) are modeled with set and linear constraints. A simple key, saying that data values determine a-nodes, is a linear constraint $x_a = \sum_{a \in S} z_S$, and an inclusion constraint saying that data values of a-nodes occur also as data values of b-nodes is $V(a) \cap \overline{V(b)} = \emptyset$. Similarly, one can handle *denial constraints*, often used in dealing with inconsistent data. An example of a denial constraint is saying that the same data value cannot appear in both an a-node and a b-node; this is expressible as $V(a) \cap V(b) \neq \emptyset$.

Our result implies that the satisfiability problem for key, inclusion, and denial constraints with respect to an automaton is solvable in NP. Note however that to express a key as a linear constraint one needs exponentially many (in Σ) variables z_S, while we can compactly encode keys simply by letters involved in them, requiring $\log |\Sigma|$ bits instead. It turns out that this does not change the bound for keys and inclusion constraints; our proof can easily be adjusted to show:

Corollary 6.1. *The satisfiability problem for key (encoded by* $\log |\Sigma|$ *bits) and inclusion constraints with respect to an automaton is solvable in* NP.

Proof. Let $\mathcal{A}$ be an automaton over the alphabet Σ and let $\mathcal{C}$ be a collection of keys and inclusion constraints, where elements of $\mathcal{C}$ are written as $V(a) \mapsto a$ (for keys) and $V(a) \subseteq V(b)$ (for inclusion constraints). Let $\Sigma = \{a_1, \ldots, a_\ell\}$.

Our algorithm to decide whether there exists a data tree $t \in \mathcal{L}(\mathcal{A})$ such that $t \models \mathcal{C}$ works as follows.

(1) Construct the existential Presburger formula $\varphi_{\mathcal{A}}(x_1, \ldots, x_\ell)$ for the automaton $\mathcal{A}$ according to Lemma 4.2.
(2) Let $\varphi_{\mathcal{C}}(x_1, \ldots, x_\ell)$ be the formula: $\exists v_1 \cdots \exists v_\ell$

$$\bigwedge_i v_i \leq x_i \quad \wedge \quad \bigwedge_i (v_i = 0 \leftrightarrow x_i = 0)$$

$$\wedge$$

$$\Big(\bigwedge_{V(a_i)\mapsto a_i\in\mathcal{C}} v_i = x_i\Big) \wedge \Big(\bigwedge_{V(a_i)\subseteq V(a_j)\in\mathcal{C}} v_i \leq v_j\Big).$$

(3) Let $\varphi_{\mathcal{A},\mathcal{C}}(x_1,\ldots,x_\ell) := \varphi_{\mathcal{A}}(x_1,\ldots,x_\ell) \wedge \varphi_{\mathcal{C}}(x_1,\ldots,x_\ell)$.
Test the satisfiability of $\varphi_{\mathcal{A},\mathcal{C}}(x_1,\ldots,x_\ell)$.

Note that here we do not use the variables z_S's.

We claim that for each data tree t, $t \in \mathcal{L}(\mathcal{A})$ and $t \models \mathcal{C}$ if and only if $\varphi_{\mathcal{A},\mathcal{C}}(\#a_1(t),\ldots,\#a_\ell(t))$ holds.

We start with the "only if" part. Let $t \in \mathcal{L}(\mathcal{A})$ and $t \models \mathcal{C}$. That $\varphi_{\mathcal{A}}(\#a_1(t),\ldots,\#a_\ell(t))$ follows from Lemma 4.2. To show that $\varphi_{\mathcal{C}}(\#a_1(t),\ldots,\#a_\ell(t))$ holds, we let the witnesses for each v_i as the cardinality $|V_t(a_i)|$, the number of data values found in the a_i-nodes in t. Then, it is straightforward to show that $\varphi_{\mathcal{C}}(\#a_1(t),\ldots,\#a_\ell(t))$ holds.

Now we show the "if" part. Suppose $\varphi_{\mathcal{A},\mathcal{C}}(n_1,\ldots,n_\ell)$ holds. By Lemma 4.2, there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that for each $a_i \in \Sigma$, $n_i = \#a_i(t)$. Let $(m_1,\ldots,m_\ell)$ be the witness for $(v_1,\ldots,v_\ell)$ that $\varphi_{\mathcal{C}}(x_1,\ldots,x_\ell)$ holds. We assign the values $1,\ldots,m_i$ as data values for the a_i-nodes in t such that $V_t(a_i) = \{1,\ldots,m_i\}$, for each $a_i \in \Sigma$. Such assignment is always possible since $m_i \leq \#a_i(t)$. That the keys and inclusion constraints in $\mathcal{C}$ are satisfied follows immediately from the constraints $v_i = x_i$ and $v_i \leq v_j$, respectively. □

This extends the results of [Arenas et al. 2008; Fan and Libkin 2002] which showed an NP bound for keys and a special form of inclusions (whose right-hand-sides are keys as well); but in addition our proof is much more streamlined compared to the proofs there.

We give a couple of remarks here comparing the results in our paper to other types of XML constraints.

Remark 6.2. It is easy to extend these results to more complex constraints studied in the XML context. For example, consider key constraints given by regular expressions over Σ. Such a constraint for a regular expression e, is satisfied in a tree t if nodes reachable from the root by following a path from e are uniquely determined by their data values. These constraints, common in XML schema specifications, are easily described by our formalism: one simply marks the nodes with states of an automaton for e, and uses the tree automaton $\mathcal{A}$ to ensure that the marking is correct.

Remark 6.3. The proof can also be easily extended to the satisfiability of conditional-XPath formulae with key and inclusion constraints on the selected nodes. It is known that a conditional-XPath formula can be converted to a query automaton [Libkin and Sirangelo 2010]. Recall that a query automaton is an automaton that in addition to accepting/rejecting a tree, also outputs a set of nodes [Neven and Schwentick 2002]. The translation of [Libkin and Sirangelo 2010] works in ExpTime. By viewing those selected nodes as nodes that carry special symbols, the results in the previous sections hold immediately also for the XPath formulae.

Remark 6.4. Finally, notice that there are different kinds of key and inclusion constraints, called *relative* constraints, studied in [Buneman et al. 2002; Arenas et al. 2008]. In short, a relative key constraint states that any two a-nodes, sharing a common ancestor labeled with b, carry different data values, while a relative inclusion constraint states that for every a-node, which has an ancestor labeled with c, there exists a b-node, with the same ancestor, that carries the same data value. The satisfaction problem of DTD and relative constraints is already proved to be undecidable in [Arenas et al. 2008].

6.2. Two-variable logic

As mentioned already, our main result does *not* follow from the decidability of the two-variable existential monadic second-order logic over data trees [Bojanczyk et al. 2009]. We

now explain the precise relationship between the two formalisms. When we talk about logics over data trees, we view them as structures

$$t \;=\; \langle D, E_{\downarrow}, E_{\rightarrow}, \{a(\cdot)\}_{a\in\Sigma}, \sim\rangle, \tag{2}$$

which extend unranked trees with the binary predicate $\sim$ interpreted as $u \sim u' \Leftrightarrow \mathrm{val}_t(u) = \mathrm{val}_t(u')$, where u and u' are nodes of the tree.

The sentences of the logic $\exists\mathrm{MSO}^2$ are of the form $\exists X_1 \ldots \exists X_m\ \psi$, where ψ is an FO formula over the vocabulary extended with the unary predicates $X_1, \ldots, X_m$ that uses only two variables, x and y. It is known that every MSO sentence that does not mention data values is equivalent to an $\exists\mathrm{MSO}^2$ sentence.

The unary key constraint $V(a) \mapsto a$ can be expressed with the formula: $\forall x\forall y\ (a(x) \wedge a(y) \wedge x \sim y \rightarrow x = y)$. The inclusion constraint $V(a) \subseteq V(b)$ can be expressed with the formula: $\forall x\exists y\ (a(x) \rightarrow b(y) \wedge x \sim y)$. The denial constraint $V(a) \cap V(b) = \emptyset$ can be expressed with the formula: $\forall x\forall y\ (a(x) \wedge b(y) \rightarrow \neg(x \sim y))$.

It was shown in [Bojanczyk et al. 2009] that every $\exists\mathrm{MSO}^2$ formula over data trees is equivalent to a formula

$$\exists X_1 \ldots \exists X_k(\chi \wedge \bigwedge_i \varphi_i \wedge \bigwedge_j \psi_j)$$

where

(1) χ describes a behavior of an automaton that can make "local" data comparisons (i.e., whether a data value in a node is equal/not equal the data value of its parent, left- or right-sibling);
(2) each φ_i is of the form $\forall x\forall y(\alpha(x) \wedge \alpha(y) \wedge x \sim y \rightarrow x = y)$, where α is a conjunction of labeling predicates, X_k's, and their negations; and
(3) each ψ_j is of the form $\forall x\exists y\ \alpha(x) \rightarrow (x \sim y \wedge \alpha'(y))$, with α, α' as in item 2.

If we extend the alphabet to $\Sigma \times 2^k$ so that each label also specifies the family of the X_i's the node belongs to, then formulae in items 2 and 3 can be encoded by constraints.

— Formulae in item 2 become conjunctions of keys and denial constraints over the extended alphabet. That is, it becomes a formula

$$\forall x\forall y(\bigvee_{a\in\Sigma'} a(x) \wedge \bigvee_{a\in\Sigma'} a(y) \wedge x \sim y \rightarrow x = y),$$

for some $\Sigma' \subseteq \Sigma \times 2^k$, which is equivalent to a is a key for each $a \in \Sigma'$, as well as $V(a) \cap V(b) = \emptyset$, for every $a, b \in \Sigma'$ and $a \neq b$.
— Formulae in item 3 become

$$\forall x\exists y(\bigvee_{a\in\Sigma'} a(x) \rightarrow x \sim y \wedge \bigvee_{a\in\Sigma''} a(y)),$$

for some $\Sigma', \Sigma'' \subseteq \Sigma \times 2^k$, which is equivalent to generalized inclusion constraints of the form

$$\bigcup_{a\in\Sigma'} V(a) \subseteq \bigcup_{b\in\Sigma''} V(b),$$

or, equivalently $\bigcup_{a\in\Sigma'} V(a) \cap \bigcap_{b\in\Sigma''} \overline{V(b)} = \emptyset$.

Hence, [Bojanczyk et al. 2009] and our results imply the following.

Corollary 6.5.

— (corollary of [Bojanczyk et al. 2009]) *Satisfiability of* $\exists\text{MSO}^2$ *formulae over data trees is equivalent to satisfiability of keys, denial constraints, and generalized inclusions constraints with respect to an automaton with local data comparisons.*
— (corollary of Theorem 4.1) *Satisfiability of keys, denial constraints, and generalized inclusions constraints with respect to an automaton is solvable in* NP*.*

While our main result and the decidability of $\exists\text{MSO}^2$ are incomparable, in essence we subsume $\exists\text{MSO}^2$ *minus the local data comparison* constraints. More precisely, by local data comparison constraints we mean those of the form:

$$\forall x \forall y((a(x) \wedge b(y) \wedge \varepsilon(x,y)) \rightarrow \delta(x,y)),$$

where $\varepsilon(x,y)$ is either "x is the parent of y" or "x is the right-sibling of y"; and $\delta(x,y)$ is either "$x \sim y$" or "$\neg(x \sim y)$."

Our formalism is suitable for stating properties that do not involve such local data comparisons. For example, the property "all data values are the same" can be expressed as $V(a) \cap \overline{V(b)} = \emptyset$ for each $a, b \in \Sigma$; and $z_\Sigma = 1$, where Σ denotes the alphabet. The property "all data values are different" can also be expressed in our formalism as follows: $V(a) \cap V(b) = \emptyset$ for each $a, b \in \Sigma$; and $x_a = \sum_{a \in S} z_S$ for each $a \in \Sigma$.

Note that our proof is conceptually simpler than the proof of [Bojanczyk et al. 2009] that goes via more than a dozen reductions. Unlike [Bojanczyk et al. 2009], we fail to capture local data comparisons in automata; on the other hand, we add many properties (e.g., linear constraints) which are not even expressible in MSO.

7. CONCISE REPRESENTATIONS OF THE SATISFIABILITY PROBLEM

Recall that we measure the size of the linear data constraints $\mathbf{A}\bar{v} \geq \mathbf{b}$ as the sum of sizes of $\mathbf{A}$ and $\mathbf{b}$, with numbers represented in binary.

This could be a rather inefficient way of representing linear constraints. Since the number of variables z_S in the constraints is $2^{|\Sigma|}-1$, we may achieve a more compact representation if only a few of those variables are used in the constraints. Namely, we can safely disregard all the zero-columns in $\mathbf{A}$, and keep only the columns that correspond to variables actually used in constraints. This representation can be exponentially smaller than the full representation of the constraints (since Σ is a part of the input, we cannot achieve a smaller reduction even if there are no linear constraints).

This is what we mean by *concise representation*. Consider the corresponding problem Concise-SAT$(\mathcal{A}, \mathcal{C})$, which is the same as the SAT$(\mathcal{A}, \mathcal{C})$ problem before, except that we use a concise representation of linear constraints.

It is a very common phenomenon in complexity theory that going to concise representation increases the complexity by an exponent; in fact doing so is a common way of getting NExpTime-complete problems from NP-complete problems. Of course given a concise representation of constraints, we can always convert it into the usual representation in at most exponential time, and then apply Theorem 4.1. This immediately tells us that Concise-SAT is in NExpTime, and it is tempting to think that Concise-SAT is NExpTime-complete.

However, this is not the case. Quite surprisingly, the concise representation does *not* increase the complexity of the problem. To show this, we need to design the decision procedure in a much more careful way.

Theorem 7.1. *The problem* Concise-SAT$(\mathcal{A}, \mathcal{C})$ *is solvable in* NP*.*

We now indicate where the proof of Theorem 4.1 falls short when we have concise representations. First, the transformation from set to linear constraints is polynomial in the number of variables z_S, i.e., $O(2^{|\Sigma|})$. This did not cause problems before, but now we may not have all the variables z_S, so the input may be of the size $O(|\Sigma|^k)$ for a fixed k. Then

the algorithm for eliminating set constraints becomes exponential. Second, the introduction of new variables v_a for $\sum_{a \in S \subseteq \Sigma} z_S$ used in the proof may likewise induce an exponential blow-up when considering concise representation.

The main aim is to show that *there exists a subset $\mathcal{Z} \subseteq 2^\Sigma$ of polynomial size such that there exists a tree $t \in \mathcal{L}(\mathcal{A})$ and $t \models \mathcal{C}$ if and only if there exists a tree $t' \in \mathcal{L}(\mathcal{A})$ and $t' \models \mathcal{C}$ and $[S]_{t'} = \emptyset$, for all $S \notin \mathcal{Z}$.* For this we introduce another extension of the ILP problem. In the following three subsections we present the proof of Theorem 7.1.

7.1. Proof of Theorem 7.1

Let Σ be a finite alphabet and $\mathcal{C}$ is a collection of set and linear constraints. In the following we say that a term $\tau \in \mathcal{C}$ if and only if $\mathcal{C}$ contains a set constraint of the form $\tau = \emptyset$ or $\tau \neq \emptyset$. Similarly we say that a variable $z_S \in \mathcal{C}$ if and only if there is a linear data constraint in $\mathcal{C}$ that uses z_S. We denote by $\Psi_{\text{lin}}(\mathcal{C})$ the set of linear data constraints found in $\mathcal{C}$.

Definition 7.2. [$\mathcal{C}$-functions] Given an alphabet Σ and a collection $\mathcal{C}$ of data constraints, a $\mathcal{C}$-function is a function $\mathcal{F}$ from $\Sigma \cup \{\tau \mid \tau \in \mathcal{C}\} \cup \{z_S \mid z_S \in \mathcal{C}\}$ to 2^Σ such that:

(C1) for each $a \in \Sigma$, either $\mathcal{F}(a) = \emptyset$ or $a \in \mathcal{F}(a)$;
(C2) for each $z_S \in \mathcal{C}$, either $\mathcal{F}(z_S) = \emptyset$ or $\mathcal{F}(z_S) = S$;
(C3) for each constraint $\tau \neq \emptyset \in \mathcal{C}$, we have $\mathcal{F}(\tau) \in \mathbb{S}(\tau)$;
(C4) for each constraint $\tau = \emptyset \in \mathcal{C}$, we have $\mathcal{F}(\tau) = \emptyset$ and $Im(\mathcal{F}) \cap \mathbb{S}(\tau) = \emptyset$;

where $Im(\mathcal{F})$ denotes the image of $\mathcal{F}$, and $\mathbb{S}(\tau)$ was defined in Subsection 4.2.

The intuition of $\mathcal{F}$ is such that $Im(\mathcal{F})$ is the desired set $\mathcal{Z}$. Given a collection $\mathcal{C}$ of data constraints and a $\mathcal{C}$-function $\mathcal{F}$, we denote by $\Psi(\mathcal{C}, \mathcal{F})$ the system obtained from $\mathcal{C}$ by adding the following constraints to $\Psi_{\text{lin}}(\mathcal{C})$:

$z_S \geq 1$	for each $S \in Im(\mathcal{F}) - \emptyset$
$x_a = 0$	for each $a \in \Sigma$ such that $\mathcal{F}(a) = \emptyset$
$z_S = 0$	for each $z_S \in \mathcal{C}$ such that $\mathcal{F}(z_S) = \emptyset$
$\sum_{a \in S \in Im(\mathcal{F}) - \emptyset} z_S \leq x_a$	for each $a \in \Sigma$;

Notice that the size of $\Psi(\mathcal{C}, \mathcal{F})$ is polynomial in the size of both $\mathcal{C}$ and the alphabet Σ.

In the rest of the proof, all instances of ILP we refer to are instances over the variables x_a, z_S, v_a.

Definition 7.3 (*ILP under $\mathcal{C}$-condition*). An instance of ILP problem under $\mathcal{C}$-condition is given by an instance Ψ of ILP together with a collection $\mathcal{C}$ of data constraints. We say that it has a non-negative solution if there exists a $\mathcal{C}$-function $\mathcal{F}$ such that the instance of ILP given by Ψ and $\Psi(\mathcal{F}, \mathcal{C})$ has a non-negative solution.

We will now state the two main lemmas from which Theorem 7.1 follows immediately.

LEMMA 7.4. *Checking whether an instance of ILP under $\mathcal{C}$-condition has a non-negative solution is in* NP.

LEMMA 7.5. *Given an automaton $\mathcal{A}$ and a set $\mathcal{C}$ of data constraints in concise representation, one can construct, in polynomial time, an instance of ILP with $\mathcal{C}$-condition so that there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that $t \models \mathcal{C}$ if and only if the instance of ILP with $\mathcal{C}$-conditions has a non-negative solution.*

7.2. Proof of Lemma 7.4

First we need to prove the following result:

LEMMA 7.6. *Given an alphabet Σ, a collection $\mathcal{C}$ of linear data constraints, and a function $\mathcal{F}$ from $\Sigma \cup \{\tau \mid \tau \in \mathcal{C}\} \cup \{z_S \mid z_S \in \mathcal{C}\}$ to 2^Σ one can decide in polynomial time if $\mathcal{F}$ is a $\mathcal{C}$-function.*

Proof. To check that $\mathcal{F}$ is a $\mathcal{C}$-function, one has to check that conditions (C1)–(C4) in definition 7.2. Conditions (C1) and (C2) are easy to check in polynomial time. The fact that Conditions (C3) and (C4) can be checked in polynomial time follows directly from the following claim:

CLAIM 1. *Given a term τ and a set $S \subseteq \Sigma$, one can decide in linear time if $S \in \mathbb{S}(\tau)$.*

PROOF. The proof is similar to the one in Subsection 4.2. In Subsection 4.2 we show how to convert a boolean formula to a term. Here we show how to convert a term into a boolean formula.

For each $a \in \Sigma$, we associate a boolean variable P_a. And for each term τ, we associate a boolean formula φ_τ over the variables P_a as follows:

— if $\tau = V(a)$, then $\varphi_\tau = P_a$;
— if $\tau = \tau_1 \cup \tau_2$, then $\varphi_\tau = \varphi_{\tau_1} \vee \varphi_{\tau_2}$.
— if $\tau = \tau_1 \cap \tau_2$, then $\varphi_\tau = \varphi_{\tau_1} \wedge \varphi_{\tau_2}$;
— if $\tau = \overline{\tau}_1$, then $\varphi_\tau = \neg\varphi_{\tau_1}$.

A set $S \subseteq \Sigma$ defines a Boolean assignment ξ_S, where the variable $\xi_S(P_a) = \mathsf{true}$ if and only if $a \in S$, for each $a \in \Sigma$.

It is a rather straightforward induction to show that the boolean formula φ_τ evaluates to true under the assignment ξ_S if and only if $S \in \mathbb{S}(\tau)$. The construction of φ_τ can be done in time linear in the length τ, and so is the evaluation of φ_τ under ξ_S. This completes the proof of our claim. □

We can now prove Lemma 7.4.

PROOF. (of Lemma 7.4) Note the size of the system $\Psi(\mathcal{C}, \mathcal{F})$ is polynomial in $|\Sigma|$ and $\Psi(\mathcal{C})$, and hence if a solution to some $\Psi(\mathcal{C}, \mathcal{F})$ exists, there is one of size polynomial in $|\Sigma|$ and $\Psi(\mathcal{C})$ due to [Papadimitriou 1981]. Hence for an NP algorithm we simply guess $\mathcal{F}$ and a solution to $\Psi(\mathcal{C}, \mathcal{F})$. Both guesses are of polynomial size, and then we check that $\mathcal{F}$ is a $\mathcal{C}$-function, build $\Psi(\mathcal{C}, \mathcal{F})$ and verify that the solution is correct. These three tasks can be done in polynomial time. □

7.3. Proof of Lemma 7.5

Recall that $(\mathcal{A}, \mathcal{C})$ is satisfiable if and only if there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that $t \models \mathcal{C}$. The same meaning applies to $(\mathcal{A}, \Psi(\mathcal{C}, \mathcal{F}))$, where $\mathcal{F}$ is a $\mathcal{C}$-function.

The following Lemma 7.7 immediately implies Lemma 7.5.

LEMMA 7.7. *Let $\mathcal{A}$ be an automaton and $\mathcal{C}$ a collection of data constraints. Then, $(\mathcal{A}, \mathcal{C})$ is satisfiable if and only if there exists a $\mathcal{C}$-function $\mathcal{F}$ such that $(\mathcal{A}, \Psi(\mathcal{C}, \mathcal{F}))$ is satisfiable.*

PROOF. The "if" direction is trivial. If there exists a tree t that satisfies $(\mathcal{A}, \Psi(\mathcal{C}, \mathcal{F}))$, then the same tree also satisfies $(\mathcal{A}, \mathcal{C})$.

We now show the "only if" part. That is, if $(\mathcal{A}, \mathcal{C})$ is satisfiable, then there exists a $\mathcal{C}$-function $\mathcal{F}$ and a data tree $t_\mathcal{F}$ such that $[S]_{t_\mathcal{F}} = \emptyset$ for all $S \notin Im(\mathcal{F})$ and $t_\mathcal{F} \models (\mathcal{A}, \Psi(\mathcal{C}, \mathcal{F}))$.

Consider a tree t that satisfies $(\mathcal{A}, \mathcal{C})$. Define a $\mathcal{C}$-function $\mathcal{F}$ as follows:

— For each $a \in \Sigma$, if the label a does not appear in t, then define $\mathcal{F}(a) = \emptyset$. Otherwise, define $\mathcal{F}(a) = S_a$, where S_a is such that $a \in S_a$ and $[S_a]_t \neq \emptyset$. Such a set exists as at least one node in t is labeled by a.
— For each $z_S \in \mathcal{C}$, if $[S]_t = \emptyset$ define $\mathcal{F}(z_S) = \emptyset$. Otherwise, define $\mathcal{F}(z_S) = S$.

— For each constraint $\tau \neq \emptyset \in \mathcal{C}$, define $\mathcal{F}(\tau) = S_\tau$ where $S_\tau \in \mathbb{S}(\tau)$ and $[S_\tau]_t \neq \emptyset$. Such a set exists as $t \models \tau \neq \emptyset$.
— For each constraint $\tau = \emptyset \in \mathcal{C}$, define $\mathcal{F}(\tau) = \emptyset$.

It is rather straightforward to show that $\mathcal{F}$ is a $\mathcal{C}$-function.

We now build from t and $\mathcal{F}$ a data tree $t_\mathcal{F}$ that satisfies $(\mathcal{A}, \Psi(\mathcal{C}, \mathcal{F}))$ and such that $[S]_{t_\mathcal{F}} = \emptyset$ for all $S \notin Im(\mathcal{F})$.

The tree $t_\mathcal{F}$ is obtained from t by rearranging the data value in such a way that $[S]_{t_\mathcal{F}} = [S]_t$ for $S \in Im(\mathcal{F})$ and $[S]_{t_\mathcal{F}} = \emptyset$ for all other set S. The domain of t and $t_\mathcal{F}$ coincide and the labeling function ℓab is the same as well. We assign data values in $t_\mathcal{F}$ as follows:

— for each node u such that $\mathrm{val}_t(u) \in [S]_t$ for some set $S \in Im(\mathcal{F})$, we set $\mathrm{val}_{t_\mathcal{F}}(u) = \mathrm{val}_t(u)$;
— for each node u such that $\mathrm{val}_t(u) \in [S]_t$ and $S \notin Im(\mathcal{F})$, we let $\mathrm{val}_{t_\mathcal{F}}(u)$ be an arbitrary data value from $[\mathcal{F}(\ell ab(u))]_t$.

By this construction, $[\mathcal{F}(\ell ab(u))]_t$ is not empty and $[S]_{t_\mathcal{F}} = \emptyset$ for all $S \notin Im(\mathcal{F})$. As the tree (the data-free part) $t_\mathcal{F}$ has the same label as t, it is accepted by the automaton $\mathcal{A}$. Moreover, we have $\#a(t) = \#a(t_\mathcal{F})$ for each $a \in \Sigma$ and $[S]_t = [S]_{t_\mathcal{F}}$ for each $z_S \in \mathcal{C}$. Since the tree t satisfies $\mathcal{C}$, the tree $t_\mathcal{F}$ satisfies the linear data constraints from $\Psi_{\mathrm{lin}}(\mathcal{C})$.

It remains to show that it satisfies the following additional constraints:

(i). $z_S \geq 1$ for each $S \in Im(\mathcal{F}) - \{\emptyset\}$.

(ii). $x_a = 0$ for each $a \in \Sigma$, where $\mathcal{F}(a) = \emptyset$.

(iii). $z_S = 0$ for each $S \in \mathcal{C}$, where $\mathcal{F}(z_S) = \emptyset$.

(iv). $$\sum_{a \in S \in Im(\mathcal{F}) - \{\emptyset\}} z_S \leq x_a \quad \text{for each } a \in \Sigma.$$

Constraints (i)-(iii) are ensured by the construction of $\mathcal{F}$. As in the proof of Theorem 4.1, in the formula $\chi(\bar{v}, \bar{x}, \bar{z})$ and the fact that $t \in \mathcal{L}(\mathcal{A})$ and $t \models \mathcal{C}$, we have:

$$\sum_{a \in S \subseteq \Sigma} |[S]_t| \leq \#t(a) \qquad \text{for each } a \in \Sigma$$

By construction of the tree $t_\mathcal{F}$, we have $\#t_\mathcal{F}(a) = \#t(a)$ for each $a \in \Sigma$ and $|[S]_{t_\mathcal{F}}| \leq |[S]_t|$ for each non-empty set $S \subseteq \Sigma$. Thus, the constraint (iv) is satisfied by $t_\mathcal{F}$. □

8. CONVERTING AUTOMATA TO PRESBURGER FORMULAE

To make our proof completely algorithmic, we spell out the translation from an automaton to a Presburger formula defining Parikh images of trees, used as a black box (Lemma 4.2) in the proof of Theorem 4.1. This is the algorithm that we use in our implementation described in Section 9. We also present an algorithm that constructs a tree accepted by the original automaton from a solution to the corresponding Presburger formula.

We recall that the Parikh image of a tree t over the alphabet $\Sigma = \{a_l, \ldots, a_\ell\}$ is the ℓ-tuple $\mathsf{Parikh}(t) = (\#a_1(t), \ldots, \#a_\ell(t)) \in \mathbb{N}^\ell$, and the Parikh image of a tree language L is $\mathsf{Parikh}(L) = \{\mathsf{Parikh}(t) \mid t \in L\} \subseteq \mathbb{N}^\ell$.

PROPOSITION 8.1. *There is a quadratic time algorithm that, given an unranked tree automaton $\mathcal{A}$ over $\Sigma = \{a_l, \ldots, a_\ell\}$, returns a formula*

$$\varphi_\mathcal{A}(x_1, \ldots, x_\ell) \;=\; \exists \bar{y}\ \psi(\bar{x}, \bar{y})$$

of at most quadratic size such that

— *if* $t \in \mathcal{L}(\mathcal{A})$*, then* $\varphi_{\mathcal{A}}(\#a_1(t), \ldots, \#a_\ell(t))$ *holds; and conversely,*
— *if* $\varphi_{\mathcal{A}}(n_1, \ldots, n_\ell)$ *holds, then there exists a tree* $t \in \mathcal{L}(\mathcal{A})$ *such that* $\#a_1(t) = n_1, \ldots, \#a_\ell(t) = n_\ell$

and α *is a conjunction of formulae of the form:*

— $\mathbf{A}(\bar{x}, \bar{y}) \geq \mathbf{b}$*, where all the entries of* $\mathbf{A}$ *and* $\mathbf{b}$ *are either* 0 *or* 1 *or* -1*;*
— *formulae* $(v = 0 \vee v' \geq 1)$ *where* v, v' *are variables among* $\bar{x}, \bar{y}$*; and*
— *disjunctions* $\bigvee_i (v_i \geq 1 \wedge v_i' = 1)$*, where* v_i*'s and* v_i'*'s are variables among* $\bar{x}, \bar{y}$*.*

Moreover, from every solution $(n_1, \ldots, n_\ell)$ *and witness tuple* $\bar{m}$ *such that* $\alpha(n_1, \ldots, n_\ell, \bar{m})$ *holds, we can construct effectively a tree* $t \in \mathcal{L}(\mathcal{A})$ *such that* $\mathsf{Parikh}(t) = (n_1, \ldots, n_\ell)$*.*

8.1. Proof of Proposition 8.1

The general outline is as follows: we first replace an automaton by an ECFG – Extended Context-Free Grammar (Proposition 8.2), and then by a CFG of a special form, which we call *simple* CFG (Proposition 8.3). We then show the construction of the Presburger formula for such simple CFGs (Proposition 8.4). The first two reductions are standard. The crucial one is the last one.

Recall that an ECFG is a CFG in which the right-hand sides of productions are regular expressions. Formally, an ECFG over the alphabet $(\Gamma \cup \Lambda)$ of nonterminals Γ, with a distinguished symbol r for the root, and terminals Λ is $\mathfrak{G} = (\Gamma, \Lambda, \Delta)$, where Δ assigns to each symbol $a \in \Gamma$ a regular expression over $(\Gamma \cup \Lambda) - \{r\}$. The set of trees of $\mathfrak{G}$ is denoted by $\mathcal{T}(\mathfrak{G})$. That is, an unranked tree t is in $\mathcal{T}(\mathfrak{G})$ if its root is labeled r, for each node v labeled $a \in \Gamma$ with children $u \cdot 0, \ldots, u \cdot (n-1)$, the word of their labels, i.e., $\ell ab_t(u \cdot 0) \cdots \ell ab_t(u \cdot (n-1))$, is in the language of $\Delta(a)$, and each node labeled with $b \in \Lambda$ is a leaf.

The first reduction is stated as a proposition below.

Proposition 8.2. *Given an automaton* $\mathcal{A}$ *with the set* Q *of states over alphabet* Σ*, one can construct, in quadratic time, an ECFG* $\mathfrak{G} = (\Gamma, \Sigma - \{r\}, \Delta)$ *with* $\Gamma = Q \times \Sigma \cup \{r\}$ *such that the following holds.*

(1) *For all tree* $t \in \mathcal{L}(\mathcal{A})$*, there exists a tree* $t' \in \mathcal{T}(\mathfrak{G})$ *such that for all* $a \in \Sigma$*,* $\#a(t) = \#a(t')$*.*
(2) *Vice versa, for all tree* $t' \in \mathcal{T}(\mathfrak{G})$*, there exists a tree* $t \in \mathcal{L}(\mathcal{A})$ *such that for all* $a \in \Sigma$*,* $\#a(t) = \#a(t')$*.*

Moreover, every tree $t' \in \mathcal{T}(\mathfrak{G})$ *can be converted effectively into a tree* $t \in \mathcal{L}(\mathcal{A})$*.*

Proof. Let $\mathcal{A} = (\Sigma, Q, \delta, F)$ be an automaton. For a regular expression α over the alphabet Q, we define the expression $\overline{\alpha}$ as follows.

— If $q \in Q$, then $\overline{q} = \bigcup_{a \in \Sigma} (q, a)$.
— $\overline{\beta \cup \gamma} = \overline{\beta} \cup \overline{\gamma}$; $\overline{\beta\, \gamma} = \overline{\beta}\, \overline{\gamma}$, and $\overline{(\beta)^*} = (\overline{\beta})^*$.

The desired ECFG $\mathfrak{G} = (Q \times \Sigma \cup \{r\}, \Sigma - \{r\}, \Delta)$ is defined as follows.

— $\Delta(r) = \bigcup_{q_f \in F} \overline{\delta(q_f, r)}$.
— For every $(q, a) \in Q \times \Sigma$, $\Delta((q, a)) = \overline{\delta(q, a)}\; a$.

Transforming a tree $t' \in \mathcal{T}(\mathfrak{G})$ to a tree $t \in \mathcal{L}(\mathcal{A})$ is straightforward.

(1) Delete every nodes in t' labeled with $\Sigma \cup \{r\}$; and it results in an accepting run of $\mathcal{A}$, which is a tree over the alphabet $Q \times \Sigma$.
(2) Project this accepting run to the alphabet Σ. The resulting tree is the desired t.

This completes the proof of Proposition 8.2. □

Next, we define a *simple CFG* as $\mathcal{G} = (\Gamma, \Lambda, \Delta)$ with a designated terminal symbol $\lambda \in \Lambda$. For each $a \in \Gamma$, $\Delta(a)$ is of the form: b, or $b \cdot c$, or $b|c$, or λ, where $b, c \in (\Gamma \cup \Lambda) - \{r\}$. We denote the set of parse trees of $\mathcal{G}$ by $\mathcal{T}(\mathcal{G})$. Note that trees in $\mathcal{T}(\mathcal{G})$ can have only unary or binary branching. We make the standard assumption that all symbols in Γ are reachable from the root symbol r. If a CFG has some unreachable symbols, they can be eliminated without affecting the set $\mathcal{T}(\mathcal{G})$.

The second reduction is stated as proposition below.

PROPOSITION 8.3. *Given an ECFG $\mathfrak{G} = (\Gamma, \Lambda, \Delta)$, one can construct, in linear time, a simple CFG $\mathcal{G} = (\Gamma', \Lambda \cup \{\lambda\}, \Delta')$ such that the following holds.*

(1) *For all tree $t \in \mathcal{T}(\mathfrak{G})$, there exists a tree $t' \in \mathcal{T}(\mathcal{G})$ such that for all $a \in \Lambda$, $\#a(t) = \#a(t')$.*
(2) *Vice versa, for all tree $t' \in \mathcal{T}(\mathcal{G})$, there exists a tree $t \in \mathcal{T}(\mathfrak{G})$ such that for all $a \in \Lambda$, $\#a(t) = \#a(t')$.*

Moreover, every tree $t' \in \mathcal{T}(\mathcal{G})$ can be converted effectively into the tree $t \in \mathcal{T}(\mathfrak{G})$.

PROOF. Let $\mathfrak{G} = (\Gamma, \Lambda, \Delta)$ be an ECFG. We inductively construct the simple CFG $\mathcal{G} = (\Gamma', \Lambda \cup \{\lambda\}, \Delta')$, for some alphabet $\Gamma' \supseteq \Gamma$, as follows. Start with $\Gamma' = \Gamma$ and $\Delta' = \Delta$ where all symbols of Γ' are "unmarked." While Γ' contains an unmarked symbol a, do the following:

(1) If $\Delta'(a)$ is a symbol from $\Gamma' \cup \Lambda$ or λ, then mark a.
(2) If $\Delta'(a) = \alpha \ \mathsf{op} \ \beta$, where op is $\cdot$ or $|$, then
   — add new unmarked symbols A^a_α and A^a_β to Γ;
   — redefine $\Delta'(a)$ as $A^a_\alpha \ \mathsf{op} \ A^a_\beta$;
   — define $\Delta'(A^a_\alpha) = \alpha$ and $\Delta'(A^\alpha_\beta) = \beta$
   — mark a.
(3) If $\Delta(a) = \alpha^*$, then
   — add unmarked symbols $A^a_{\alpha^*}$ and A^a_α to Γ' and a marked symbol λ^a_α;
   — redefine $\Delta'(a)$ as $A^a_{\alpha^*}|\lambda^a_\alpha$;
   — define $\Delta'(A_{\alpha^*}) = A^a_{\alpha^*}|A^a_\alpha$ and $\Delta'(A_\alpha) = \alpha$ as well as $\Delta'(\lambda^a_\alpha) = \lambda$;
   — mark a.

To see that this procedure terminates in polynomial time, we define the size $\|\alpha\|$ of a regular expression as 1 for a single symbol, and by the rules $\|\alpha \ \mathsf{op} \ \beta\| = \|\alpha\| + \|\beta\| + 1$, and $\|\alpha^*\| = \|\alpha\| + 3$ (to ensure that $\|\alpha^*\| > \|A^a_{\alpha^*}|A^a_\alpha\|$). Given an ECFG $\mathfrak{G}$, let $f_{\mathfrak{G}} : \mathbb{N} \to \mathbb{N}$ be a partial function mapping each n to the number of regular expressions $\Delta(a)$ of size n for unmarked symbol a. For two partial functions $f, g : \mathbb{N} \to \mathbb{N}$ with finite support, we let $f \prec g$ be defined lexicographically, i.e., for the largest n on which they differ, either $f(n)$ is undefined, or $f(n) < g(n)$. Clearly this is a well-ordering, and each step of the algorithm, which passes from a grammar $\mathfrak{G}'$ to $\mathfrak{G}''$, guarantees $f_{\mathfrak{G}''} \prec f_{\mathfrak{G}'}$, showing that the maximum number of steps is polynomial in the total size of the regular expressions in Δ. This establishes the termination, and especially the polynomial upper bound of the procedure. Moreover, the result of the procedure is a simple CFG.

Transforming every tree $t' \in \mathcal{T}(\mathcal{G})$ to a tree $t \in \mathcal{T}(\mathfrak{G})$ is straightforward and very similar to the standard transformation between ranked trees and unranked trees. Whenever there exists a node $u \in \mathsf{Dom}(t')$ labeled with a symbol from $\Gamma' - \Gamma$, do the following.

(1) Let v be the parent of u.
(2) If u has one child: u', then delete u in t' and make u' as a child of v. All other node and edges are left untouched.

(3) Similarly, if u has two children: u_1, u_2, then delete u in t' and make u_1, u_2 as children of v. All other nodes and edges are left untouched.

When all nodes are labeled with symbols from Γ, the tree is in $\mathcal{T}(\mathfrak{G})$. This completes the proof of Proposition 8.3. □

The last reduction is stated as proposition below.

PROPOSITION 8.4. *Given a simple CFG $\mathcal{G} = (\Gamma, \Lambda \cup \{\lambda\}, \Delta)$, where $\Lambda = \{a_1, \ldots, a_\ell\}$, one can construct, in linear time, an existential Presburger formula $\varphi_{\mathcal{G}}(x_1, \ldots, x_\ell) = \exists \bar{y} \psi(\bar{x}, \bar{y})$ such that for every tree t, $t \in \mathcal{T}(\mathcal{G})$ if and only if $\varphi_{\mathcal{G}}(\#a_1(t), \ldots, \#a_\ell(t))$ holds. Moreover, from every solution $(k_1, \ldots, k_n)$ and $\bar{m}$ such that $\psi(n_1, \ldots, n_\ell, \bar{m})$ holds, we can construct effectively a tree $t \in \mathcal{T}(\mathcal{G})$ such that* $\mathsf{Parikh}(t) = (n_1, \ldots, n_\ell)$.

We devote the rest of this subsection to the proof of Proposition 8.4. We need a new notation here. For a tree t over the alphabet $\Gamma \cup \Lambda \cup \{\lambda\}$, we define a *directed* graph $G_t = (V_t, E_t)$, where the set of vertices is $V_t = \Gamma \cup \Lambda \cup \{\lambda\}$; and for every $a, b \in \Gamma \cup \Lambda \cup \{\lambda\}$, there is an edge $(a, b) \in E_t$ if there exists a node in t labeled with b and whose parent is labeled with a. If a symbol a does not appear in the tree t, then it is an isolated vertex in G_t.

The main idea is to prove that a tree $t \in \mathcal{T}(\mathcal{G})$ if and only if the quantities defined below:

(1) $n_a = \#a(t)$, for each $a \in \Gamma \cup \Lambda \cup \{\lambda\}$;
(2) $n_{a\downarrow b}$ is the number of b-nodes whose parents in t is labeled with a;
(3) δ_a is the length of *some* path from the root r to the symbol a in the graph G_t,

satisfy the following relations:

(1) $n_a = \sum_{b \in \Gamma \cup \Lambda} n_{b\downarrow a}$, for each $a \in \Gamma \cup \Lambda \cup \{\lambda\}$.
(2) — $n_a = n_{a\downarrow b} + n_{a\downarrow c}$, if $\Delta(a) = b|c$.
— $n_a = n_{a\downarrow b} = n_{a\downarrow c}$, if $\Delta(a) = b \cdot c$,
— $n_a = n_{a\downarrow b}$, if $\Delta(a) = b$,
(3) $\delta_r = 0$;
(4) for each $a \in \Gamma \cup \Lambda \cup \{\lambda\}$ and $a \neq r$,

$$\delta_a = -1 \leftrightarrow n_a = 0 \quad \wedge \bigvee_{n_{b\downarrow a} \neq 0 \text{ and } \delta_b \neq -1} \delta_a = \delta_b + 1$$

Note that by default, we set $\delta_a = -1$, if there is no path from the root to the symbol a in the graph G_t, which means that the symbol a does not appear in t.

Then, the construction of the desired formula $\varphi_{\mathcal{G}}$ is straightforward. It uses the variables x_a's, y_a's and $x_{a\downarrow b}$'s, for all $a \in \Gamma \cup \Lambda$ and b appears in $\Delta(a)$. The intended meaning of each variable is as follows: x_a is for n_a; $x_{a\downarrow b}$ is for $n_{a\downarrow b}$; y_a is for δ_a.

The formula $\varphi_{\mathcal{G}}$ is the conjunction of the following:

— $x_r = 1$;
— $x_a = \sum_{b \in \Gamma \cup \Lambda} x_{b\downarrow a}$ for each $a \in \Gamma \cup \Lambda$;
— $x_a = x_{a\downarrow b} = x_{a\downarrow c}$ for each $\Delta(a) = b \cdot c$;
— $x_a = x_{a\downarrow b} + x_{a\downarrow c}$ for each $\Delta(a) = b|c$;
— $x_a = x_{a\downarrow b}$ for each $\Delta(a) = b$;
— $y_r = 0$;
— for each $a \in \Gamma \cup \Lambda \cup \{\lambda\}$,

$$(y_a = -1 \leftrightarrow x_a = 0) \quad \wedge \bigvee_{a \text{ appears in } \Delta(b)} y_a = y_b + 1 \wedge x_{b\downarrow a} \neq 0 \wedge y_b \neq -1.$$

The total number of variables x_a's and $x_{a\downarrow b}$'s and y_a's is linear in the size of Δ. We do not need the variables $x_{a\downarrow b}$'s, if b does not appear in $\Delta(a)$.

By existentially quantifying all the variables $x_{a\downarrow b}$'s and y_a's, we can then view $\varphi_{\mathcal{G}}$ as an existential Presburger formula with x_a's as the free variables.

Proposition 8.4 follows immediately from the lemma below.

LEMMA 8.5. *Let $\mathcal{G} = (\Gamma, \Lambda, \Delta)$ be a simple CFG. The formula $\varphi_{\mathcal{G}}(\bar{n})$ holds – where $(\bar{n}) = (n_a)_{a\in\Gamma\cup\Lambda}$ and the witnesses for $x_{a\downarrow b}$'s and y_a's are: $x_{a\downarrow b} = n_{a\downarrow b} \in \mathbb{N}$, and $y_a = d_a \in \mathbb{N}$, for $a, b \in \Gamma \cup \Lambda$ – if and only if there exists a tree $t \in \mathcal{T}(\mathcal{G})$ such that*

(1) $n_a = \#a(t)$ for each $a \in \Gamma \cup \Lambda$,
(2) $n_{a\downarrow b}$ is the number of b-nodes whose parents are a-nodes, and
(3) d_a is the length of some path from the root r to the symbol a in the graph G_t.

PROOF. From the definition of $\Psi(\mathcal{G})$, the "if" part is immediate. We prove the other implication. Let $\bar{n} = (n_a)_{a\in\Sigma}$ such that $\varphi_{\mathcal{G}}(\bar{n})$ holds. Let $n_{a\downarrow b}$ be the witness for $x_{a\downarrow b}$ for $a, b \in \Gamma \cup \Lambda$, and d_a for y_a, for $a \in \Gamma \cup \Lambda$. Let $\tilde{G} = (\tilde{V}, \tilde{E})$ be a directed graph where the set $\tilde{V}$ of nodes is $\Gamma \cup \Lambda$ and the set $\tilde{E}$ of edges is defined as: $(a, b) \in \tilde{E}$ if and only if $n_{a\downarrow b} \neq 0$.

We shall construct a tree $t \in \mathcal{T}(\mathcal{G})$ that satisfies (1) and (2) and that $G_t = \tilde{G}$. First, we construct a *directed* graph $G = (V, E)$ with the following properties.

(i) For each $a \in \Gamma \cup \Lambda$, there are exactly n_a nodes labeled a.
(ii) For each $a, b \in \Gamma \cup \Lambda$, there are exactly $n_{a\downarrow b}$ edges going from an a-node to a b-node.
(iii) There is exactly one node labeled r and it has no incoming edges (the root node).
(iv) All nodes, except the root node, have exactly one incoming edge.
(v) For all nodes, outgoing edges conform to Δ. That is, for each $a \in \Gamma$, the outgoing edges from a-nodes are as follows: if $\Delta(a) = b \cdot c$, there are exactly two outgoing edges: one to a b-node and one to a c-node; if $\Delta(a) = b|c$, there is exactly one outgoing edge going to a node labeled by b or c; and if $\Delta(a) = b$, there is exactly one outgoing edge that goes to a b-node.

Procedure 1 shows the construction of the graph G. Since $n_r = 1$, there is only one root node in G. Properties (i)-(v) follow directly from the construction and the constraints given in $\Psi(\mathcal{G})$.

If G were a tree, we would be done: membership in $\mathcal{T}(\mathcal{G})$ would follow from (v), property (1) from (i), and property (2) from (ii) and (v). Therefore, to finish the proof of Lemma 8.5, we show Claim 2 and Claim 4 below.

CLAIM 2. *A connected directed graph $G = (V, E)$ that satisfies (i)-(v) is a tree.*

PROOF. From Properties (iii) and (iv), we can see that the graph G satisfies the equation $|E| = |V| - 1$. If we forget about orientation, this equation implies that a connected graph is a tree [West 2001]. The root (the r-labeled node) gives the tree a unique orientation; we must show that it is the same one as the one in G. For this, consider any path from the root to a leaf in the tree, and suppose one edge has an orientation different from G. Let (u, u') be the first such edge; that is, in G we have an edge (u', u). This cannot be the first edge of the path, as the root has no incoming edge in G. Hence u has a parent u' in the oriented tree, and the edge (u'', u) has the same orientation in both the oriented tree and in G. But this tells us that u has two incoming edges, which contradicts (iv). □

We shall use Claim 3 to prove Claim 4.

CLAIM 3. *In the directed graph $\tilde{G}$, a node a is connected from the root symbol r if and only if $d_a \neq -1$, or equivalently, $n_a \neq 0$. Moreover, d_a is the length of some path from the root symbol r to a, if $d_a \neq -1$.*

ALGORITHM 1: Construct Graph $G = (V, E)$

Data: A solution $(\bar{n}_a, \bar{n}_{a\downarrow b})_{a,b\in\Gamma\cup\Lambda}$ to the formula $\Psi(\mathcal{G})$.
Result: A tree $t \in \mathcal{T}(\mathcal{G})$ such that for every $a \in \Gamma \cup \Lambda$
— $n_a = \#a(t)$;
— $n_{a\downarrow b}$ = the number of b-nodes in t whose parents are a-nodes.

begin
  The set V consists of $\sum_{a\in\Gamma\cup\Lambda} n_a$ nodes.
  for $a \in \Gamma \cup \Lambda$ **do**
    Label n_a number of nodes with a.
  end
  $E := \emptyset$.
  for $a \in \Gamma$ **do**
    Let $u_1, \ldots, u_{n_a}$ be the a-nodes.
    if $\Delta(a) = b \cdot c$ **then**
      Let $n = n_a = n_{a\downarrow b} = n_{a\downarrow c}$.
      Pick a sequence $u'_1, \ldots, u'_n$ of n distinct b-nodes with no incoming edges in E.
      Pick a sequence $u''_1, \ldots, u''_n$ of n distinct c-nodes with no incoming edges in E.
      $E := E \cup \{(u_i, u'_i), (u_i, u''_i)\}_{i=1,\ldots,n}$.
    end
    if $\Delta(a) = b \cup c$ **then**
      Pick a sequence $u'_1, \ldots, u'_{n_{a\downarrow b}}$ of $n_{a\downarrow b}$ distinct b-nodes with no incoming edges in E.
      Pick a sequence $u''_1, \ldots, u''_{n_{a\downarrow c}}$ of $n_{a\downarrow c}$ distinct c-nodes with no incoming edges in E.
      $E := E \cup \{(u_i, u'_i)\}_{i=1,\ldots,n_{a\downarrow b}} \cup \{(u_{n_{a\downarrow b}+j}, u''_j)\}_{j=1,\ldots,n_{a\downarrow c}}$.
    end
    if $\Delta(a) = b$ **then**
      Pick a sequence $u'_1, \ldots, u'_{n_{a\downarrow b}}$ of $n_{a\downarrow b}$ distinct b-nodes with no incoming edges in E.
      $E := E \cup \{(u_i, u'_i)\}_{i=1,\ldots,n_{a\downarrow b}}$.
    end
  end
end

PROOF. The proof is by straightforward induction on the value d_a. The base case, $d_a = 0$, is trivial as it means $a = r$. The induction hypothesis is that for each node a with $d_a = k \neq -1$ is connected from the root symbol r by a path of length k.

Suppose b is a node such that $d_b = k + 1$. By the construction of $\varphi_{\mathcal{G}}$, there exists a node a such that $n_{a\downarrow b} \neq 0$ and $d_a = k$. By the induction hypothesis, a is connected from the root symbol r with a path of length k, and by the construction of $\tilde{G}$, there exists an edge from a to b. Thus, our claim holds. □

CLAIM 4. *From a directed graph $G = (V, E)$ that satisfies (i)-(v), one can compute in polynomial time a connected directed graph $G' = (V, E')$ that also satisfies (i)-(v).*

PROOF. The idea is to change a few edges in G in order to connect all components to the connected component that contains the r-node. We first observe the following. Suppose G consists of several connected components: $G_0, G_1, \ldots, G_l$, where G_0 is the component that contains the root node. Then, there exist a node u in G_0 and a node u' in one of $G_1, \ldots, G_l$ such that u and u' are labeled by the same symbol from Σ.

By Claim 3, if $d_a \neq -1$ (thus, $n_a \neq 0$), the symbol a is connected to the root symbol r in $\tilde{G}$, and that d_a is the length of some path from r to a in $\tilde{G}$. So, for every symbol a that appears in G, there exists a sequence of symbols $b_0, b_1, \ldots, b_j$, respectively, where

— $b_0 = r$,

— $b_l = a$, and
— for each $i = 0, \ldots, j-1$, $n_{b_i \downarrow b_{i+1}} \neq 0$.

If the symbol a does not appear in G_0, then there are a node u in G_0 and a node u' in one of $G_1, \ldots, G_l$ such that both u and u' are labeled with the same symbol $b_i \in \{b_1, \ldots, b_l\}$.

Let G_1 be the component that contains the node u'. By (v), the node u' has as many children as u (and it has at least one child as it is not labeled by λ).

If u and u' have one child each, then let u_1 and u_2 be their respective children. We can then connect G_0 and G_1 by replacing the edges (u, u_1) and (u', u_2) with (u, u_2) and (u', u_1). If u and u' have two children each, then let u_1, u_1' and u_2, u_2' be their respective children. We can then connect G_0 and G_1 by replacing the edges $(u, u_1), (u, u_1')$ and $(u', u_2), (u', u_2')$ with $(u, u_2), (u, u_2')$ and $(u', u_1), (u', u_1')$.

It is straightforward to see that after such edge replacement the graph still satisfies properties (i)-(v), and each edge replacement reduces the number of connected components, so eventually this algorithm produces a tree t that satisfies (i)-(v). Moreover, the numbers $n_{a \downarrow b}$ do not change during the process, thus, $G_t = \tilde{G}$. □

This completes the proof of Lemma 8.5.

9. IMPLEMENTATION OF THE SATISFIABILITY ALGORITHM

There is strong empirical evidence that many NP-complete *reasoning* tasks are feasible in practice [Malik and Zhang 2009]. To check whether the same applies to the reasoning tasks studied here, we have implemented one version of the main algorithm, using an industrial-strength Presburger solver Z3 [de Moura and Bjørner 2008] (more on it soon). In this section we give a brief report on our experimental results.

The SMT (Satisfiability Modulo Theories) solver Z3 is an automated satisfiability checker for many-sorted first-order logic with built-in theories, including support for quantifiers [de Moura and Bjørner 2008]. For existential formulae (which is our case), Z3 acts as a decision procedure giving a satisfying assignment when a formula is satisfiable.

We have implemented the satisfiability algorithm for the case when the schema is given by a simple DTD. Specifically, we have implemented the following:

— The translation of simple DTDs (see Section 8) into Presburger formulae $\varphi_{\mathcal{G}}$, as described in the proof of Proposition 8.4.
— The translation of key and inclusion constraints into Presburger formulae, as described in the proof of Corollary 6.1.
— The translation of set constraints into Presburger formulae as described in Section 7.

We then used the Z3 solver to check satisfiability of the formulae produced by these translations. To test the translations, we use the following:

— Two parameterized families of DTDs: a family F_u of DTDs that are unsatisfiable (due to the presence of recursion), and a family F_s of satisfiable DTDs.
— A parameterized family of K keys and inclusion constraints over documents that conform to DTDs from F_s.
— A parameterized family C of set and linear constraints (obtained from key, inclusion, and denial constraints) over documents that conform to DTDs from F_s.

We measure the following:

— The time needed to translate the instance of our satisfiability problem into a formula that can be fed to Z3.
— The number of variables used in the formula fed to Z3.
— The time needed for the solver Z3 to check satisfiability of the formula.

We have run all experiments on a machine with Intel Core 2 Duo processor operating at 2.4GHz, with 2GB of memory. In all the cases we have run the experiments several times and report the average time (the deviation was never high though). For all the cases below, we provide a few sample figures that are sufficient to indicate that for sizes typical for schemas and constraints, satisfiability can be checked very efficiently. The non-round numbers of rules and constraints are due to the nature of the parameterized families we chose: for example, in the first family of parameterized DTDs below, we deal with DTDs D_n, $n > 0$, such that the number of rules in D_n is $5n - 1$. We then run the algorithm for $n = 10, \ldots, 100$; the table below reports results for $n = 10, 30, 70$, and 100.

We first report some sample results for unsatisfiable DTDs from F_u. The number of rules in DTDs is the same as the number of element types.

# of rules	# of variables	Translation time (s)	Z3 time (s)
49	181	0.01	0.02
149	901	0.09	0.11
349	1261	0.14	0.15
499	1783	0.15	0.16

We now report sample results for satisfiable DTDs from F_s. Notice that Z3 time increases, most likely due to the computation of a satisfying assignment.

# of rules	# of variables	Translation time (s)	Z3 time (s)
34	132	0.02	0.04
154	612	0.06	0.43
301	1200	0.1	1.78

Then, we consider DTDs from F_s together with keys and inclusion constraints from K.

# of rules	# of constraints	# of variables	Translation time (s)	Z3 time (s)
34	23	144	0.01	0.04
154	103	664	0.04	0.25
214	143	924	0.06	0.63
301	201	1403	0.1	3.1

Finally, we report results for DTDs from F_s and set and linear constraints from C.

# of rules	# of constraints	# of variables	Translation time (s)	Z3 time (s)
34	42	154	0.01	0.05
154	153	714	0.04	0.73
214	214	994	0.06	1.7
304	303	1504	0.1	4.52

The conclusion we can draw from these results is that the approach is indeed feasible. Note that we are dealing with schemas and constraints, so we have tested DTDs with up to 500 rules, and a significant number of constraints, exceeding 300, and the total translation and Z3 time has never exceeded 5s.

10. CONCLUSIONS

We have studied the consistency problem of set and linear constraints with respect to regular tree languages given by tree automata. This problem is motivated by many reasoning and static analysis tasks arising in the context of XML.

We have proved the decidability of our formalism, and established an NP upper bound. We provided a much simpler proof than those in the literature, which allows us to extend the result to more complex reasoning tasks and different constraint representations. The key technique is the encoding of the reasoning tasks as instances of integer linear programming (or existential Presburger formulae).

We have provided explicit algorithms for all the subtasks, and experimented with encoding them via an existing SMT solver, showing promising results for DTDs with several hundred rules together with several hundreds of constraints. In all the cases it was a matter of seconds to complete the reasoning tasks.

Given these promising initial results, we intend to expand our implementation and build a fully-fledged system for static analysis of XML schemas and constraints.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their careful reading and constructive comments.

REFERENCES

Alon, N., Milo, T., Neven, F., Suciu, D., and Vianu, V. 2003. XML with data values: typechecking revisited. *J. Comput. Syst. Sci. 66,* 4, 688–722.

Arenas, M., Fan, W., and Libkin, L. 2008. On the complexity of verifying consistency of XML specifications. *SIAM J. Comput. 38,* 3, 841–880.

Arenas, M. and Libkin, L. 2008. XML data exchange: consistency and query answering. *Journal of the ACM 55,* 2.

Björklund, H., Martens, W., and Schwentick, T. 2008. Optimizing conjunctive queries over trees using schema information. In *Mathematical Foundations of Computer Science*. Springer, 132–143.

Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., and Segoufin, L. 2011. Two-variable logic on data words. *ACM Transactions on Computational Logic 12,* 4.

Bojanczyk, M., Muscholl, A., Schwentick, T., and Segoufin, L. 2009. Two-variable logic on data trees and XML reasoning. *Journal of the ACM 56,* 3.

Bouyer, P., Petit, A., and Thérien, D. 2001. An algebraic characterization of data and timed languages. In *CONCUR*. Springer, 248–261.

Buneman, P., Davidson, S., Fan, W., Hara, C., and Tan, W.-C. 2002. Keys for XML. *Computer Networks 39,* 5.

Calvanese, D., Giacomo, G. D., Lenzerini, M., and Vardi, M. 2009. An automata-theoretic approach to regular XPath. In *Database Programming Languages*. 18–35.

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Löding, C., Lugiez, D., Tison, S., and Tommasi, M. 2007. *Tree Automata: Techniques and Applications*.

David, C., Libkin, L., and Tan, T. 2011. Efficient reasoning about data trees via integer linear programming. In *Int. Conf. on Database Theory*.

de Moura, L. M. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

Demri, S. and Lazic, R. 2009. Ltl with the freeze quantifier and register automata. *ACM Transactions on Computational Logic 10,* 3.

Fan, W. and Libkin, L. 2002. On XML integrity constraints in the presence of dtds. *Journal of the ACM 49,* 3.

Figueira, D. 2009. Satisfiability of downward xpath with data equality tests. In *Symp. on Principles of Database Systems*. ACM, 197–206.

Genevés, P. and Layaida, N. 2006. A system for the static analysis of XPath. *ACM Transactions on Information Systems 24,* 4, 475–502.

Givan, R., McAllester, D. A., Witty, C., and Kozen, D. 2002. Tarskian set constraints. *Information and Computation 174*, 105–131.

Jurdzinski, M. and Lazic, R. 2007. Alternation-free modal mu-calculus for data trees. In *Symp. on Logic in Computer Science*. IEEE Computer Society, 131–140.

Kaminski, M. and Tan, T. 2008. Tree automata over infinite alphabets. In *Pillars of Computer Science*. Springer, 386–423.

Kopczynski, E. and To, A. W. 2010. Parikh images of grammars: Complexity and applications. In *Symp. on Logic in Computer Science*. IEEE Computer Society.

Libkin, L. and Sirangelo, C. 2010. Reasoning about XML with temporal logics and automata. *Journal of Applied Logic 8,* 2, 210–232.

Malik, S. and Zhang, L. 2009. Boolean satisfiability: from theoretical hardness to practical success. *Communications of the ACM 52,* 8, 76–82.

Martens, W., Neven, F., and Schwentick, T. 2007. Simple off the shelf abstractions for XML schema. *SIGMOD Record 36,* 3, 15–22.

Marx, M. 2005. Conditional XPath. *ACM Transactions on Database Systems 30,* 4, 929–959.

Milo, T., Suciu, D., and Vianu, V. 2003. Typechecking for XML transformers. *J. Comput. Syst. Sci. 66,* 1, 66–97.

Neven, F. 2002. Automata, logic, and XML. In *Computer Science Logic*. Springer, 2–26.

Neven, F. and Schwentick, T. 2002. Query automata over finite trees. *Theoretical Computer Science 275,* 1-2, 633–674.

Neven, F., Schwentick, T., and Vianu, V. 2004. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic 5,* 3, 403–435.

Pacholski, L. and Podelski, A. 1997. Set constraints: a pearl in research on constraints. In *Principles and Practice of Constraint Programming*. 549–562.

Papadimitriou, C. 1981. On the complexity of integer programming. *Journal of the ACM 28*, 765–768.

Robinson, A. and Voronkov, A. 2001. *Handbook of Automated Reasoning*. MIT Press.

Schwentick, T. 2004. XPath query containment. *SIGMOD Record 33,* 1, 101–109.

Thatcher, J. 1967. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *J. Comput. Syst. Sci. 1*, 317–322.

Verma, K., Seidl, H., and Schwentick, T. 2005. On the complexity of equational horn clauses. In *Conference on Automated Deduction*. Springer, 337–352.

Vianu, V. 2001. A Web odyssey: from Codd to XML. In *Symp. on Principles of Database Systems*. ACM, 1–15.

West, D. 2001. *Introduction to Graph Theory*. Prentice Hall.